Semantic Impact and Faults in Source Code Changes: An Empirical Study

Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry
Electrical and Computer Engineering, The University of Texas at Austin
{dshao, khurshid, perry}@ece.utexas.edu

Abstract

Changes to source code have become a critical factor in fault predictions. Text or syntactic approaches have been widely used. Textual analysis focuses on changed text fragments while syntactic analysis focuses on changed syntactic entities. Although both of them have demonstrated their advantages in experimental results, they only study code fragments modified during changes. Because of semantic dependencies within programs, we believe that code fragments impacted by changes are also helpful. Given a source code change, we identify its impact by program slicing along the variable def-use chains. To evaluate the effectiveness of change impacts in fault detection and prediction, we compare impacted code with changed code according to size and fault density. Our experiment on the change history of a successful industrial project shows that: fault density in changed and impacted fragments are higher than other areas; for large changes, their impacts have higher fault density than changes themselves; interferences within change impact contribute to the high fault density in large changes. Our study suggests that, like change itself, change impact is also a high priority indicator in fault prediction, especially for changes of large scales.

1. Introduction

During software development and evolution, change is a pervasive phenomenon. Changes happen in every phase: requirements, design, implementation, and maintenance. Changes in any phase will affect the behavior of the delivered software product. Thus, changes are significantly related to software product quality and may be used effectively to predict faults.

Since source code is the ultimate definition of the behavior of a software product, changes in source code become an important factor in understanding changes in behaviors and detecting, or predicting, faults.

In studies based on source code changes, textual or syntactic approaches have been widely used. Textual analysis treats the source code as just a piece of text and builds change rules according to the textual properties of changes. For example, Hassam and Holt [9], Mockus and Weiss [16], and Nagappan and Ball [17] analyze how many lines added or deleted in a change and associate change size with faults. Syntactic analysis uses syntactic entities, such as classes/methods/fields in Zimmermann et al. [29], the number of incoming and outgoing function calls in Knab et al. [13], and the association or sequence of function calls in Breu and Zimmermann [3] and Kim et al. [12]. They extract change rules or patterns on syntactic entities and relate them to faults.

Although the advantages of textual and syntactic approaches have been demonstrated with experimental results, both approaches only focus on the code modified during changes. Text and syntax are inherently limited in clarifying the meaning, or delineating the implications, of changes.

Neither approach considers the semantic dependencies within a program. For example, introducing a statement assigning a value to a variable may adversely impact subsequent statements using the value of that variable. We believe these impacted statements (i.e., the *change impacts*) are extremely helpful in detecting or predicting faults. A semantic approach provides us with an inherently deeper understanding of both the meaning and the implications of changes [19].

To evaluate the effectiveness of change impact in fault detection and prediction, we conduct a local data-flow analysis of the change history of a successful industry project, the 5ESS™ system. For each sampled change, we identify its impact by program slicing along the variable def-use chains. We group changes according to change size and compare the size and fault density of change impacts and changes themselves across groups. In this study, we want to address three questions: for small changes and large changes,

1) Is there any difference in the size of their impacts?
2) Is there any difference in the ratio of impact size to change size?
3) Is there any difference in the faults densities in change impacts and in change themselves?

Section 2 gives an overview of the identification of semantic change impact by local data-flow analysis. The context for this study is discussed in Section 3. Section 4 presents the experimental design and its results. Validity issues are discussed in Section 5. Related research is discussed in Section 6. Finally, we summarize our study and propose future work in Section 7.

2. Semantic Impacts of Source Code Changes

To identify the impact of source code changes, we combine local data-flow analysis[1] and program slicing. Local data-flow analysis discloses the internal dependencies between semantic structures in a program while program slicing identifies semantic structures impacted by changes. Figure 1 illustrates our semantic analysis of change impact.

Given two versions (the first is the original version, the second is the changed version), we first calculate the control flow graph for each version. From each control flow graph, we identify the dependencies based

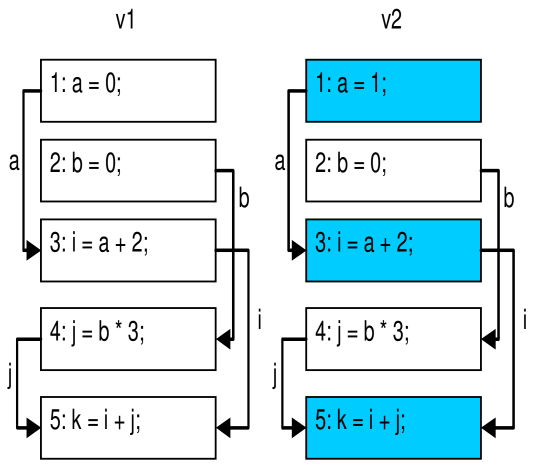

Figure 1. Semantic analysis on change: v1→v2

[1] "Local data-flow analysis" means we only analyze dependencies within the local context, not the transitive closure that includes all callee functions. See [25] for further clarification.

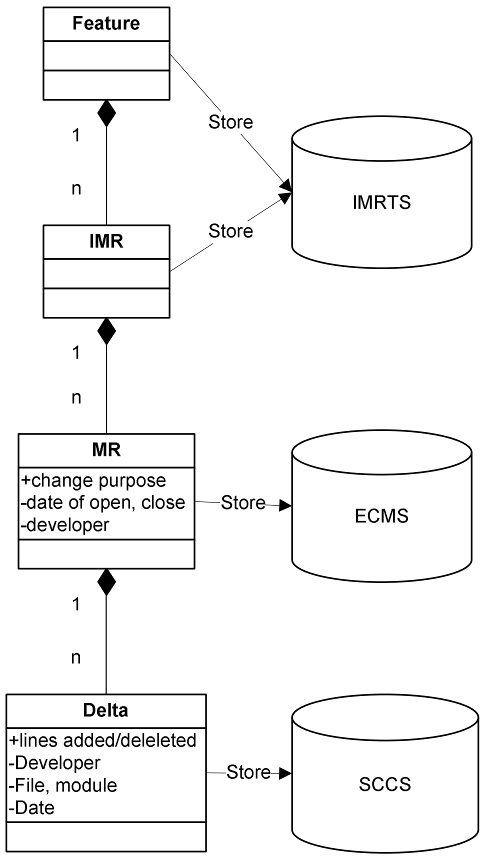

Figure 2. Change and version management repositories of 5ESS

on variable def-use pairs. Each identified local data-flow dependency is represented as a triple (*var: def, use*), where *var* is the variable on which the dependence is built, *def* is the line that defines variable *var*, and the *use* line uses the variable defined at *def* line. The dependences in version v1 are {(a: 1, 3), (b: 2, 4), (i: 3, 5), (j: 4, 5)}.

With the variable def-use dependency analysis on the two versions, the semantic impact of a change can be calculated by forward slicing from the changed lines. In this example, change v1→v2 modified Line 1 from "a = 0" to "a = 1". According to the variable def-use chains, {(a: 1, 3), (i: 3, 5)}, Line 3 and 5 are impacted. So the semantic impact of change v1→v2 is {3, 5}.

3. Study Context

The data in this study comes from the change and version management history of the Office Automation (OA) subsystem in 5ESS™, an ultra-large-scale, successful industrial project.

The 5ESS™ change history has been widely used for various studies, such as inferring change effort from configuration management database [7], automatically extracting reasons for changes [15], studying the semantic impact of parallel changes and small changes [20] [21]. Such a widely used data source increases the content validity of our study.

3.1 Change and Version Management Repositories

In 5ESS™, changes are managed in a layered hierarchy (Figure 2): Feature, Initial Modification Request (IMR), Modification Request (MR) and delta. A feature is the fundamental unit of extension to the system. Each feature is implemented by a set of IMRs that represent problems to be solved. An IMR is a change for fixing a fault, perfecting or improving some aspect of the system, or adding new features to the system. In 5ESS™, IMRs are recorded in the IMR Tracking System (IMRTS).

Each IMR is implemented by one or more MRs. Each MR represents a solution, or part of a solution, to an IMR's problem. An MR is usually owned by one developer and can represent all or part of the developer's contribution to the solution to an IMR. Information about MRs is kept in an ECMS [26] database. For each MR, ECMS records the data such as the date it was opened and closed, its developer, and a short text abstract of the change purpose.

In the context of a MR, every change made to a file was recorded as a delta in SCCS [23]. A *delta* records the actual lines added or deleted in a file. For each delta, the ECMS also record its date, the developer, and the MR it belongs to.

The data used in this study are: the lines added or deleted on source code files and the purpose for the changes. These data can be retrieved from the ECMS and SCCS system.

3.2 Change Classification

Changes can be classified based according to the purpose of the change. In our study, the purpose-based classification of MRs was derived from the work by Mockus and Votta [15]. They classified MRs according to the keywords in the textual abstract of a change. For example, if keywords like "fix," "bug," "error," and "fail" were present, the change was classified as corrective. There are four classes of changes:

- *Inspection changes*: found in code inspections.
- *Perfective changes*: changes to improve performance, make a piece of code more maintainable, or generally improve the quality of the code.
- *Adaptive changes*: changes to add new features and functionality into the system.
- *Corrective change*: changes to fix faults.

Given a change, its class is the class of the MR it belongs to. For example, a Corrective MR represents one fault and all the changes belonging to a Corrective MR are Corrective changes for that fault.

In this study, Corrective changes are used to identify faults. *Matching Corrective* changes are used as a metric to evaluate fault prediction capabilities. Details are discussed in Section 4.2.

3.3 Implementation Issues

The implementation of the data dependency calculation and program slicing is based on GrammaTech's CodeSurfer [5]. The C compiler is Visual C++ 6.0. For the variables used in def-use analysis, we count both pointer variables and non-pointer variables. For pointer analysis, we select CodeSurfer's option that distinguishes individual fields in a referenced structure and allow Codesurfer to perform pointer analysis.

4. Study and Results

To address the three questions mentioned before, we propose three hypotheses:

1) H1: Large changes impact more lines than small changes;
2) H2: Large changes have a lower impact size to change size ratio than small changes;
3) H3: For fault density of impacted code, large changes are higher than small changes; but for fault density of changed code, large changes are lower than small changes. Fault density of changed or impacted code is higher than that of the code fragments that are neither changed nor impacted by changes.

4.1. Constructs

To evaluate these three hypotheses, we need to define constructs to represent change, change impact, and fault:

Change: Change has different meanings in different contexts. In this study, change is represented as the difference in lines between the old version and the new

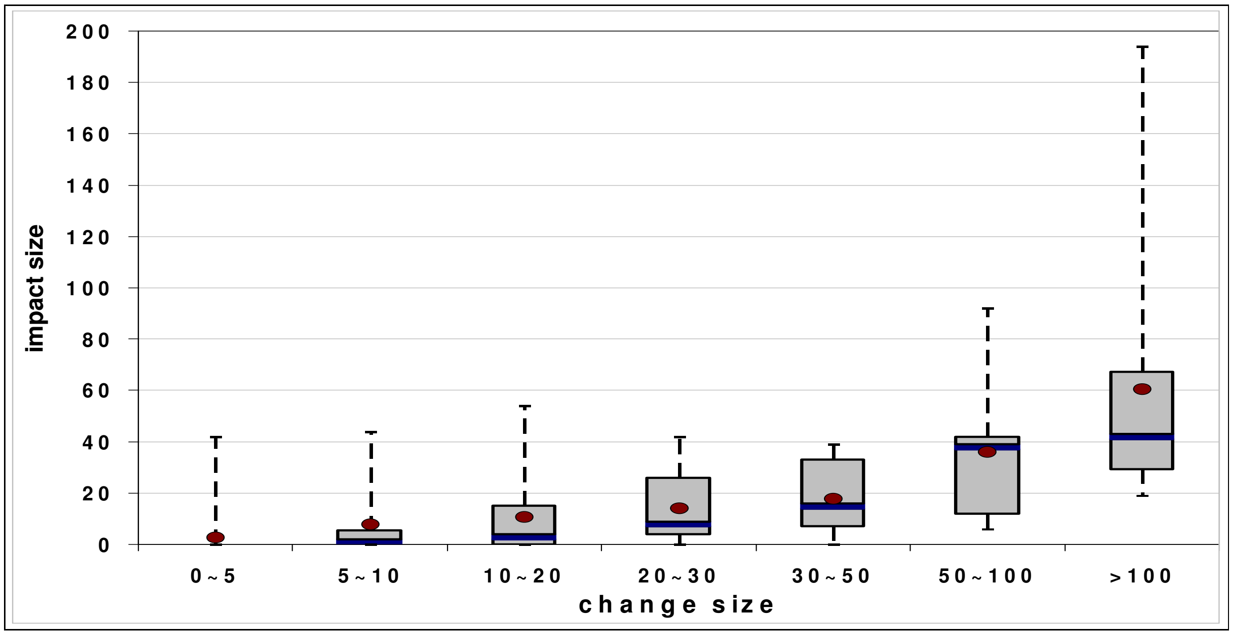

Figure 3. Distribution of impact size.

version. Given a change, *change size* is the number of the lines added, deleted, and modified in the change. But, SCCS system only records the number of lines inserted or deleted during a change. Modifications to the existing lines are tracked as old lines being replaced by new lines (delete and insert). We use the maximum of added and deleted lines as change size.

Change impact: Given a change, its impact is represented as the lines that have local data dependencies on changed lines. According to the variable def-use chains, the impact of a change is identified by forward slicing from the changed lines. *Impact size* is the number of lines in a change impact.

Fault: A fault is the collection of all lines changed in a Corrective type change.

4.2. Studies

In this study, 273 changes were sampled from four modules in 5ESS change repository. The size of each version is around 1000 lines. All the changes are grouped according to change size. For all groups, their distributions of different change purposes (adaptive, corrective, and perfective) are very similar.

To evaluate the three hypotheses, we performed three studies to compare these groups according to following three metrics: impact size, ratio of impact size to change size, and fault density.

4.2.1 Impact size

According to the sampled changes and their impacts, Figure 3 shows the distribution of impact size for each group. The x-axis is change size and the y-axis is impact size. For each group, the vertical line is the box-plot representation of the distribution of impact size within that change-size group. On each vertical line, the box shows the central 50% of the distribution. The upper and lower ends of the box mark the upper and lower quartiles. The bold dot shows the mean of the distribution. The solid horizontal line in the box shows the median of the distribution. The dashed vertical lines attached to the box indicate the tails of the distribution. The vertical lines above and below the box are the maximum and minimum. This figure shows that, with the increase of the change size, the change impact size also increases accordingly. For small changes that modified less than 5 lines, the median and mean of their impact size are 0 and 2.587413. For large changes modifying more than 100 lines, the median and mean of their impact increase to 42 and 60.5.

For all sampled changes, statistical analysis on their change size and impact size shows that they are strongly correlated. Their *correlation coefficient* is R = 0.75, with a very high significance p <= 3.24e-48. In the t-test for the significance of the coefficient, t = 18.76 with *Degrees of Freedom* = 271.

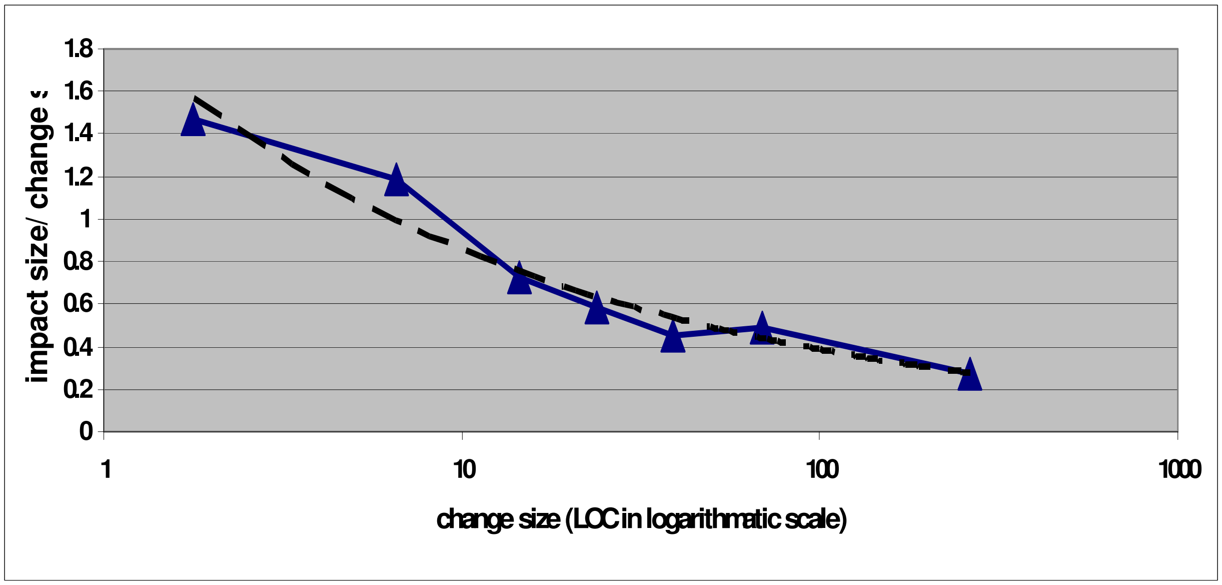

Figure 4. Ratio of impact size to change size

The results from statistical analysis and Figure 3 support hypothesis H1: large changes impact more lines than small changes.

This result supports our previous study on small changes [21]. In the change history of this subsystem of 5ESS, less than 4% of small (one-line) changes result in faults. According to the result from H1, impacts of small changes are also small. Because there is little effect on the semantics of the program, small changes are not likely to introduce faults.

4.2.2 Ratio of impact size to change size

For each group, we also calculate the ratio of impact size to change size. Figure 4 illustrates the results. The x-axis is the change size and the y-axis is the ratio of impact size to change size. The x and y coordinate of each point represents the average of change size and ratio for each group. The trend shows that, with the increase of change size, the ratio goes down quickly. For changes less than 10 lines, their impact size is larger than change size. For changes around 70 lines, the impact size is only half the change size. Statistical analysis shows a nearly linear relation between change size (on a logarithmtic scale) and the ratio of impact size to change size. Their *correlation coefficient* is $R = -0.951$, $p <= 0.00103$. In the t-test for the significance of the coefficient, $t = 6.874$ with *Degrees of Freedom* = 5

The results of our statistical analysis and Figure 4 support hypothesis H2: Large changes have lower impact size to change size ratio than small changes. The relative small impact size suggests that, to detect or predict faults in large changes, checking the change impact is cheaper than checking the change itself.

This result also discloses the internal interference phenomena of large changes. For a large change, the impacts of different changed lines highly overlap with each other. In a program, a change on one line usually touches at least one variable def-use chain. With the increase of change size, more and more variable def-use chains are touched. However, the size of change impact, the union of all lines in all touched variable def-use chains, increases very slowly. For changes less than 5 lines, the impact size is larger than change size, and their impact-change ratio is 1.47. For changes less than 100 lines and greater than 50 lines, the impact size is much smaller than change size, their ratio is only 0.49. With the increase on change size, more and more lines are shared by the impacts of different changed lines. Since more and more affected variable def-use chains overlap or converge, their interference degree becomes higher and higher. According to the trend in Figure 4, the interference within large changes is very significant.

4.2.3 Fault density

To evaluate the effectiveness of change impact in fault prediction, we compare the fault density in impacted code with changed code across the seven groups.

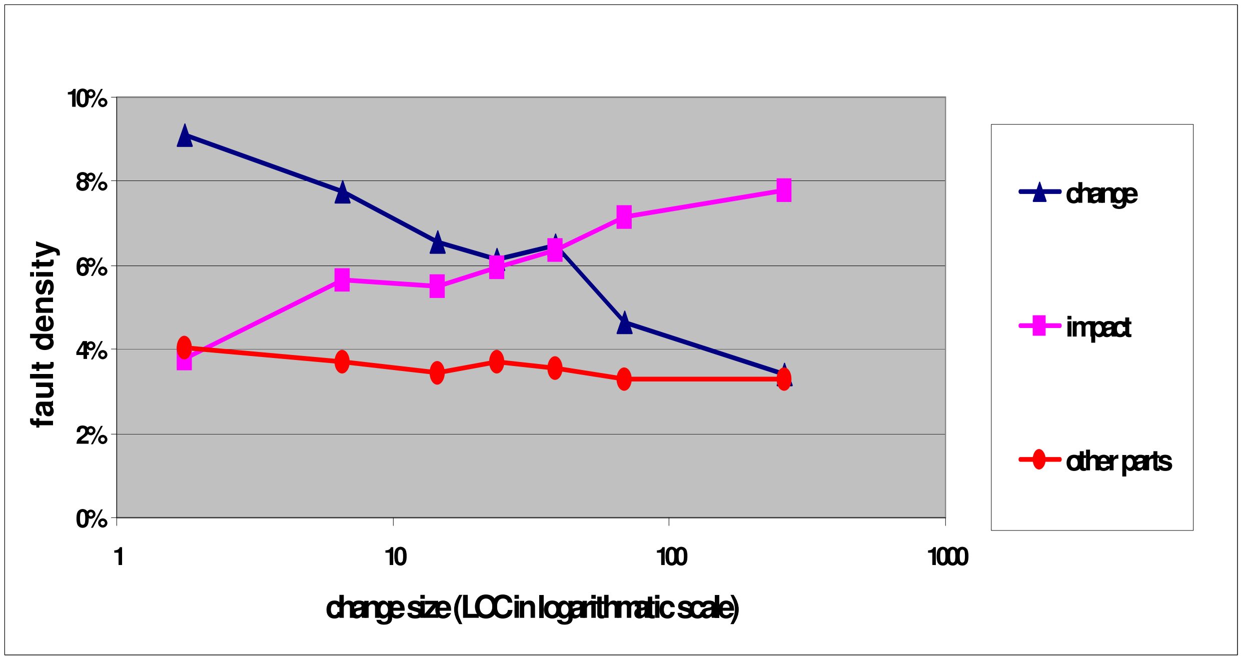

Figure 5. Fault density in change and impact.

Identify Fault in change, impact and other parts

To find faults introduced by a change, we search for Corrective MRs subsequent (in time) to this change in ECMS. Then, in SCCS, we identify faults by the code fragments changed in deltas belong to these Corrective MRs.

Given a change, whatever Corrective or not, we divide it into two sets of source code lines: added lines and deleted lines. Modified lines can be defined as added new lines and removed old lines.

For a Change *c*: Vi→Vj,

Added-lines(*c*) = {Line *n* | Line *n* exist in version Vj but not in version Vi};

Deleted-lines(*c*) = {Line *n* | Line *n* exist in version Vi but not in version Vj};

Impacted-lines(*c*) = {Line *n* | Line *n* in version Vj and Line *n* in forward slicing of change *c*}.

Other-lines(*c*) = {Line *n* | Line *n* in version Vj} - Added-lines(*c*) - Impacted-lines(*c*).

Given a Change *c* and a subsequent Corrective MR *m*, the fault in change, its impact, and other parts are defined as following:

Change-Fault-Match(*c*, *m*) = Added-lines(*c*) ∩ Deleted-lines(*m*).

Impact-Fault-Match(*c*, *m*) = Impacted-lines(*c*) ∩ Deleted-lines(*m*).

Other-Fault-Match(*c*, *m*) = Other-lines(*c*) ∩ Deleted-lines(*m*).

The intersections are calculated by textual matching source code lines.

If *Change-Fault-Match(c, m)* ≠ *Ø*, a fault is found in Change *c*. It is same while searching fault in change impact and other parts. Note that we do not capture faults of omission (that is, faults that are corrected only by adding lines); deleted lines represent changes to fix faults.

Compare fault density of change, impact and other parts

With faults identified in change, impact and other parts, we calculate fault density for each of them. For each change, the ratio of fault size to impact size is the *fault density of the impact*; the ratio of fault size to change size is the *fault density of the change itself*; the ratio of fault size to the size of these code fragments is the fault density of other parts.

Figure 5 illustrates the fault density of change impact and change itself according to size groups. The x-axis is the change size while the y-axis is fault density. With the increase of change size, the fault density in change impact increases and fault density in change itself decreases. For changes less than 5 lines, their average change size is 1.5 lines and the fault density in their change impact is only 3.7%, while fault density in change itself is about 9.1%. For changes less than 100 lines and greater than 50 lines, their average change size is 70 lines. Their fault density in change

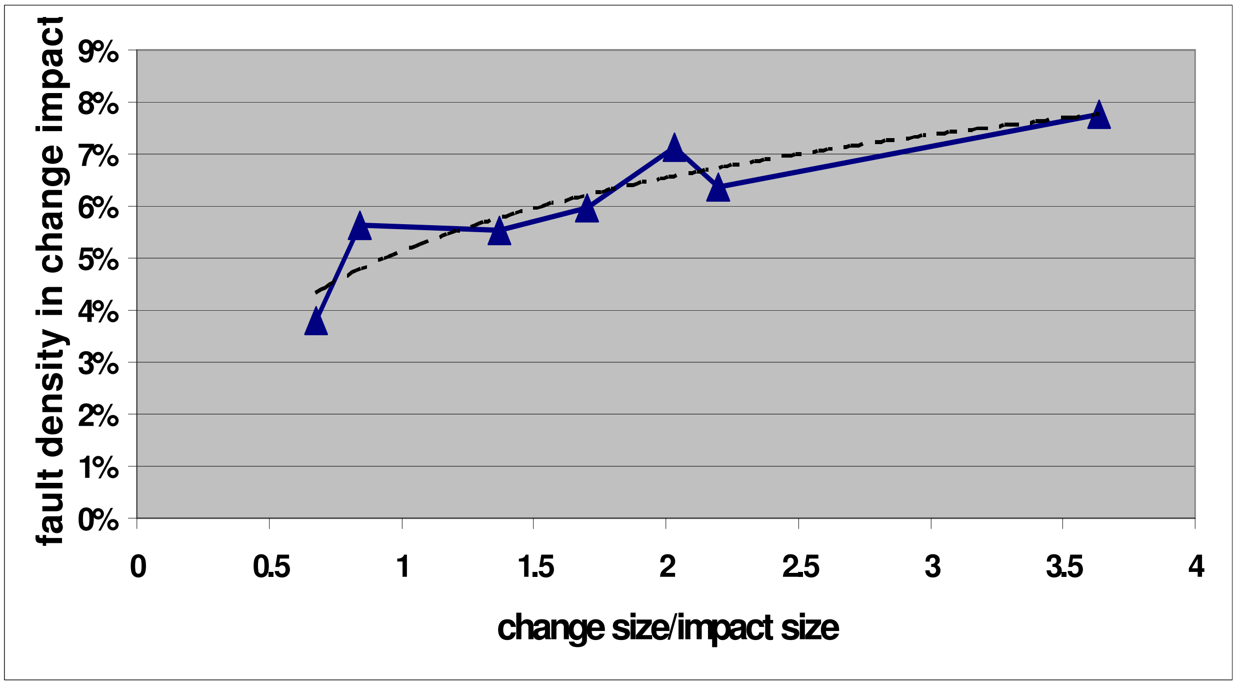

Figure 6. Relation between internal interference and fault density in change impacts.

impact increases to 7.1%. But the fault density in change itself drops to 4.6%.

Figure 5 also shows that, although the fault density in change and its impact varies according to change size, both of them are higher than fault density parts that are not changed or impacted. For all change groups, the fault densities in other parts are always around 4%. It is roughly equal to the lowest fault density of changed or impacted code of all change size groups.

We performed statistical analysis on the correlation between change size (on a logarithmic scale) and the ratio of impact fault density to change fault density. Their *correlation coefficient* is $R = 0.92$, $p <= 0.003339$. In the t-test for the significance of the coefficient, $t = 5.351$ with *Degrees of Freedom* = 5

The results of our statistical analysis and Figure 5 support hypothesis H3: for fault density in change impacts, large changes are higher than small changes; but for fault density in the change themselves, large changes are lower than small changes. The high fault density in change impact suggests that, to detect or predict faults in large changes, checking the change impact is more effective than checking the change itself.

Combining hypothesis H2 and H3, we come to a conclusion: change impacts can be complementary to changes themselves in detecting or predicting faults. For large changes, checking change impact is efficient because of its relatively small size and high density; for small changes, checking the change themselves is more efficient. Checking changes or their impacts is more efficient than checking other parts.

4.3. Result Analysis

Our study shows that large changes are dangerous, not only because of the large change scope, but also for the high fault density in their impacts. It is more severe in that change impacts are not as easily identified as changes themselves.

Because of the high internal interference found in impacts of large changes, we suggest that internal interference contributes to the fault density of change impact. In a large change, modified lines are distributed across a large scope and their impacts highly overlap with each other. It is difficult for a developer to identify all of the impacts and their high degree of interference.

Statistical analysis on the relation between internal interference and fault density in change impact also supports our suggestion. We use the ratio between change size to impact size as the metric for *internal interference*. A larger ratio value means higher internal interference. Figure 6 shows this result. The x-axis is the ratio of change size to impact size and the y-axis is the fault density in change impact. The x and y coordinate of each point represents the average of change-impact ratio and fault density of each group. The trend shows that, with the increase change-impact ratio (interference), the fault density in change impact increases. For low interference where the ratio is 0.67, the fault density is less than 4%. For high interference where the ratio is 3.6, the fault density is almost 8%. The correlation analysis between change-impact ratio and fault density shows that their *correlation coefficient* is $R = 0.877$, $p <= 0.01149$. In t-test for the significance of the coefficient, $t = 4.081$, with *Degrees of Freedom* = 5.

There might be multiple reasons for large change with small impact. For example, for a larger size change, its impact, which is a subset of the unchanged code fragment, is likely to be smaller because we only analyse change and its impact locally, within a function. However, whatever the actual reason is, the high ratio between change size to impact size indicates that large changes are more dangerous than expected.

5. Validity Analysis

To analyze the soundness of this experiment, we discuss its construct, internal, and external validity.

Construct validity concerns in our studies center around the identification of changes and faults. In this experiment, changes are identified by textual comparison and change impacts are identified by program slicing along the variable def-use chains. Both of them are widely used constructs in program analysis. Identification of changes by textual comparison can introduce false positives. For example, breaking a statement from one line into two lines is a change according to text comparison. But such changes have no syntactic or semantic affect. However, we found such changes often when we manually checked our change impact analysis results. Faults are identified by the MR classification tools [15] with reasonable reliability. Previous studies [20] [21] showed that such classification is reasonable.

The data used in our study comes from a real industrial project and can be considered completely reliable. All the data in our study comes from an original history without any artificial instrumentation or modification.

Our fault identification approach might introduce some noise. If a fault is found in the impact of a change, the faulty code should also match at least one line in the previous change that introduced the faulty code into code base. One fault is counted twice, one in change and one in change impact. Another source of noise is that a fault in the impact of a change may not be directly related to this change. We believe that this is unlikely as corrective MRs are written only for a single faults.

To improve accuracy, we group multiple changes that belong to the same MR into to one change while matching with Corrective MRs. Grouping changes according to their MR is reasonable because an MR is a logic group of changes made by one developer to solve one task (or in the case of corrective MRs, one fault).

Although the faults found in the impact of a change maybe not directly be related to this change, it is still worth checking it. First, its fault density is high and it is almost impossible for a developer to check all previous changes while submitting a new change. Second, it is helpful to find change interference. If a piece of code added by one change was impacted by another change, the two changes interfere with each other. Our previous study [24] shows that such interferences between changes are important in predicting potential faults.

With respect to internal validity, we believe that our results are consistent with our previous studies. For the semantic impact of changes, one would intuitively expect that small changes usually touch few semantic entities and affect only a small part of a program. In this study, the statistical analysis of correlation coefficients, significant levels, and t-tests shows a strong correlation between change size and change impact size. The matching between change impact and faults has already been successfully used in the previous study on change interference [24]. These consistencies are strong support for the internal validity. Thus, we do not believe there are alternative explanations for our results.

Although our study is based on the history data in a pessimistic version control system, SCCS, this approach can be easily extended to optimistic version control systems, such as Concurrent Versions System (CVS), which is widely used in open source projects. CVS can supply the same kinds of data as SCCS for our change impact analysis algorithm.

A threat to the validity of our study is that 5ESS is a very large-scale, real-time project with a large number of developers who are geographically distributed. We argue, however, that the subsystem we studied is perhaps by itself more representative of a typical large project. It is developed with programming languages (C and C++) under a UNIX environment. Both of them are very common in software developments. For a large project like 5ESS, well-trained developers and well-organized development processes, organization, and management are required.

Replicating our study is not difficult under other change or version management systems. This study needs two kinds of data: versions and delta information (lines added, deleted or changed) and change classification. Our data comes from two repositories: the version management system (SCCS) and change management system (ECMS). Like SCCS, current version management systems, such as CVS and Subversion, can support versions and delta information. To classify change according to purposes, the text explanations of current version management systems can be used. Automatic classification approach in Mockus and Votta [15] can also be applied on these new version management systems. A widely used tracking system, such as BugZilla, can also be used to classify changes, especially the corrective changes for bug-fixing. In our study, the classification is based on MRs, a group of changes made together for a same purpose. Although MRs are not supported in CVS or Subversion, there are still other approaches to group changes. Subversion supports change transactions. Although CVS does not support transactions, other effective approaches, such as sliding windows, have been used to identify transactions. With the help from current version and change management systems, our study can be easily replicated.

6. Related Work

In studies on semantic program differencing, dependence graphs and program slicing are widely used. Horwitz [10] performs semantic program differencing by partitioning the components of the two programs according to equivalent behaviors. The algorithms based on Horwitz [10] can provide high precision semantic differencing. Binkley [1] uses system dependence graph and inter-procedural slicing to identify the impacted parts of source code changes. And Yang et al. [28] increased the soundness of semantic conflict detection with semantic preserving transformations. These semantic differencing approaches are based on the comparison on the whole dependency graph. Their workload is heavy and difficult to use in practice. While performing analysis on whole programs, the false positive in one procedure will be propagated and amplified. Our change impact identification approach only focuses on local data-flow analysis. The cost is very inexpensive. Limiting our analysis scope to local context can reduce the false positive rate. Such a compromise can reduce the cost and improve the applicability in large programs. The benefits of our approach have been demonstrated in Shao et al. [24].

Dynamic slicing is also an important approach in impact analysis. Program chopping [8] can minimize possible fault inducing code fragments. Compared with static program slicing we used in this study, dynamic slicing can improve the precision for pointer analysis and reduce false positives in semantic interference detection. However executable versions are required beforehand. This imposes a significant build overhead in a large system like 5ESS. Further, 5ESS's need for a complex hardware context may make this approach virtually impossible.

Abstract-syntax-tree-based approaches were proposed by Fluri et al. [4], Raghavan et al. [22], and Yang [27]. Given two programs, their abstract syntax trees are compared with tree-matching algorithms. These approaches identify changes on syntax entities, but do not calculate impacts from changes.

Similar to our work, Jackson and Ladd [11] check the variable def-use dependency with a program. However changes are identified only by added or deleted dependencies. Impacted dependencies are not counted.

In empirical evaluation to semantic differencing algorithms, Binkley et al. [2] applied the system dependence graph and inter-procedural slicing based semantic differencing algorithm on 10 programs. On average, the identified impact is 37% less than the whole program. The relative small size of change impact supports the application of semantic differencing in practical use. However, this work does not consider the affect of change size on change impact, and does not relate change impact with faults.

The relations between change size and faults have been studied in Graves et al. [6], Hassan and Holt [9], Mockus and Weiss [16], Nagappan and Ball [17], Ostrand et al. [18], and Purushothaman and Perry [21]. The risks of large changes were also observed in these studies. But all of them are based on changes themselves, rather than impact of changes.

Local data-flow analysis was used in our previous work to detect semantic interference and predict faults [24]. However, we studied change impact and its relation to faults according to change intervals rather than change size as we have done in this study. The interference identified in our previous study is *external interference*, happening between changes. Given two changes on the same file, they have interference if their impacts have overlap. The interference found in current study is *internal interference*. It happens within the impact of a single change.

Although external interference and internal interference come from difference sources, both of them are related to faults. External interference is related to faults because of the short interval between changes. It is difficult for a developer to fully understand the impact of all the previous changes made by others in a short interval. So the interference between short interval changes is dangerous and likely to introduce faults. Internal interference is related to faults because of large change size. Modified lines are distributed across a large scope and their impacts highly overlap with each other. It is difficult for a developer to identify all of the impacts and their interferences. In large changes, the internal interference is dangerous and is very likely to introduce faults.

7. Conclusions and Future Work

One of the fundamental problems in software evolution is that changes often introduce faults. Worse, changes to fix faults often introduce new faults. Our primary goal in our empirical studies of change is to understand these issues and what factors contribute to the occurrence of faults in software evolution. We believe that semantic-based approaches offer the most cogent means both to understanding change and to detecting and/or predicting faults.

Source code change histories provide abundant resources for understanding the evolution of software systems. Based on the source code involved in changes, many studies have been performed and interesting results have been generated. However, most of them are at the textual level or syntactic level, and do not consider internal dependencies at the semantic level. To apply semantic analysis to source code change histories, we performed an empirical study to use local data-flow analysis on the history of a successful industry project. Our studies show that:

- Large changes impact, not surprisingly, more lines than small changes; but
- Large changes have smaller impact size to change size ratio than that of small changes; and
- For fault density of impacted code, large changes are higher than small changes; but for fault density of changed code, large changes are lower than small changes. Fault density in change or its impact is higher than fault density in other parts.

For large changes, our experiment found a high degree of interference within their impacts. It is difficult for a developer to identify all of the impacts and their high degree of interference. We suggest that the high degree internal interference contributes to the high fault density in the impact of large changes.

The high fault density found in impacts aggravates the danger of making large changes. For large changes, the fault density in changes themselves has already become very dangerous. Lezak et al. [14] suggest that a component should be rewritten if more than 25% of the code is changed. The high fault density in impacts intensifies the danger of large changes because change impacts are not as easily identified as the changes themselves.

The high fault density in the impact of large changes suggests some future research topics. To detect or predict more faults, efficient calculation of the semantic impact of large change becomes important. To disclose the internal mechanisms of high fault density, more semantic analysis on change impact is required, especially on large changes. Dynamic analysis techniques, such as dynamic slicing, might also be introduced to provide more precise impact identification and fault prediction.

8. Acknowledgements

We thank Harvey Siy, University of Nebraska, Omaha, for his help on the change management system of 5ESS™. This work was supported in part by NSF CISE Grant IIS-0438967.

9. References

[1] D. Binkley, "Using semantic differencing to reduce the cost of regression testing", *Proc. of IEEE International Conference on Software Maintenance (ICSM'92)*, Orlando, FL, USA, Nov. 1992, 41-50.

[2] D. Binkley, R. Capellini, L.R. Raszewski, and C. Smith, "An implementation of and experiment with semantic differencing", *Proc. of IEEE International Conference on Software Maintenance (ICSM'01)*, Florence, Italy, Nov. 2001, 82-91.

[3] S. Breu, T. Zimmermann, "Mining Aspects from Version History", *21st IEEE International Conference on Automated Software Engineering (ASE'06)*, 2006, 221-230.

[4] B. Fluri, M. Wursch, M. Pinzger, and H.C. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction", *IEEE Transactions on Software Engineering, Special Issue on Mining Software Repositories*, Vol. 33, Issue 11, Nov. 2007, 725-743.

[5] GrammaTech, Inc. http://www.grammatech.com/

[6] T.L. Graves, A.F. Karr, J.S. Marron and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, July 2000, 653-661.

[7] T.L. Graves, A. Mockus, "Inferring Change Effort from Configuration Management Databases", *Proc. of the Fifth*

International Symposium on Software Metrics, IEEE, 1998, 267-273.

[8] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops", *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, California, Nov. 2005, 263-272.

[9] A.E. Hassan, and R.C. Holt, "The top ten list: Dynamic fault prediction", *Proc. of IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, Sep. 2005, 263-272.

[10] S. Horwitz, "Identifying the Semantic and Textual Differences between Two Versions of a Program", *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'90)*, New York, June 1990, 234-245.

[11] D. Jackson, and D.A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications", *Proc. of IEEE International Conference on Software Maintenance (ICSM'94)*, Nov. 1994, pages 243-252.

[12] M. Kim, D. Notkin, D. Grossman, "Automatic Inference of Structural Changes for Matching across Program Versions", *Proc. of the International Conference on Software Engineering (ICSE'07),* 2007, 333-343.

[13] P. Knab, M. Pinzger, A. Bernstein, "Predicting defect densities in source code files with decision tree learners", *Proc. of the 2006 international workshop on Mining software repositories (ICSE'06)*, Shanghai, China, 119 - 125.

[14] M. Leszak, D.E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective", *Journal of Systems and Software* Vol. 61, Issue 3, April 2002, 173-187.

[15] A. Mockus, and L.G. Votta, "Identifying Reasons for Software Changes Using Historic Databases", *Proc. of IEEE International Conference on Software Maintenance (ICSM'00)*, San Jose, CA, USA, October. 2000, 120-130.

[16] A. Mockus, and D.M. Weiss, "Predicting risk of software changes", *Bell Labs Technical Journal* 5, 2, 2000, 169-180.

[17] N. Nagappan, and T. Ball, "Use of relative code churn measures to predict system defect density", *Proc. of International Conference on Software Engineering (ICSE'05)*, Saint Louis MO, May 2005, 284- 292.

[18] T.J. Ostrand, E.J. Weyuker, R.M. Bell, "Predicting the location and number of faults in large software systems" *IEEE Transactions on Software Engineering*, Vol. 31, Issue 4, April 2005, 340-355

[19] D.E. Perry, "Software interconnection models", *Proc. of the 9th International Conference on Software Engineering (ICSE'87)*, Monterey, CA, March, 1987, 61- 69.

[20] D.E. Perry, H.P. Siy, and L.G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study", *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 3, July, 2001, 308-337.

[21] R. Purushothaman, and D.E Perry, "Toward Understanding the Rhetoric of Small Source Code Changes", *IEEE Transactions on Software Engineering, Special Issue on Mining Software Repositories*, Vol. 31, No. 6, June 2005, 511-526.

[22] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, V. Augustine, "Dex: A Semantic Graph Differencing Tool for Studying Changes in Large Code Bases", *Proc. of IEEE International Conference on Software Maintenance (ICSM'04)*, September 2004, 188-197.

[23] M.J. Rochkind, "The Source Code Control System", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, December 1975, 364-370.

[24] D. Shao, S. Khurshid, and D.E. Perry, "Evaluation of semantic interference detection in parallel changes: an exploratory experiment", *Proc. of the 23rd IEEE International Conference. on Software Maintenance (ICSM'07)*, Paris, France, October 2007, 74-83.

[25] G.L. Thione, and D.E. Perry, "Parallel Changes: Detecting Semantic Interferences", *The 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, July 2005, 47-56.

[26] P. A. Tuscany, "Software Development environment for Large Switching Projects", *Proc. of the 6th International Conference on Software Engineering*, Tokyo, September 1992, 58-67.

[27] W. Yang, "Identifying syntactic differences between two programs", *Software - Practice & Experience*, Vol. 21, Issue 7, July 1991, 739 - 755.

[28] W. Yang, S. Horwitz, and T. Reps, "A program integration algorithm that accommodates semantics-preserving transformations", *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 3, July 1992, 310-354.

[29] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, "Mining version histories to guide software changes", *IEEE Transactions on Software Engineering*, Vol. 31, Issue 6, June 2005, 429 - 445.