Predicting Faults from Direct Semantic Interference: An Evaluative Experiment

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

Parallel developments are becoming increasingly prevalent in the building and evolution of large-scale software systems. Our previous studies of a large industrial project showed that there was a linear correlation between the degree of parallelism and the likelihood of defects in the changes. To further study the relationship between parallel changes and faults, we have designed and implemented an algorithm to detect "direct" semantic interference between parallel changes. To evaluate the analyzer's effectiveness in fault prediction, we designed an experiment in the context of an industrial project. We first mine the change and version management repositories to find sample versions sets of different degrees of parallelism. We investigate the interference between the versions with our analyzer. We then mine the change and version repositories to find out what faults were discovered subsequent to the analyzed interfering versions. We use the match rate between semantic interference and faults to evaluate the effectiveness of the analyzer in predicting faults. Our contributions in this evaluative empirical study are twofold.. First, we evaluate the semantic interference analyzer and show that it is effective in predicting faults (based on "direct" semantic interference detection) in changes made within a short time period. Second, the design of our experiment is itself a significant contribution and exemplifies how to mine software repositories rather than use artificial cases for rigorous experimental evaluations.

1. Introduction

Parallel development has become a common phenomenon in the development of large-scale software systems. Multiple developers work on the same module or program at the same time. The need for parallel development has come about for a variety of reasons:

- the size of the software systems,
- time to market also brings pressure to develop new features or new products in a very short time,
- code ownership management is too expensive,
- the increase of globalization, and
- the geographical distribution of developers.

While parallel development increases productivity, it also causes problems. When developers work in parallel, it is likely that their changes may unintentionally interfere with each other.

In our earlier work [12] [13], we delineated the phenomena of, and the problems related to, parallel changes. In a subsystem of Lucent Technologies' 5ESS™ Telephone Switching System, high degrees of parallelism happened at multiple levels. To disclose the relationship between parallel changes and faults, we studied *prima facie* conflicts at the textual level, checking the overlap between the lines changed by different developers. We found two important results: 1) 3% of the changes made within 24 hours by different developers physically overlapped each others' changes; and 2) there was a linear correlation between the degree of parallelism and the likelihood of a defect in the changes.

Our initial investigations focused on explicit syntactic conflicts. We believe that there are also conflicts at the semantic level. To explore this hypothesis, we designed a semantic interference detection algorithm [21] [22], based on data dependency analysis and program slicing.

To investigate our hypothesis as well as the effectiveness and efficiency of our algorithm, we built a semantic conflict analyzer (SCA), designed a rigorous empirical study, and executed it in the same industrial context as our previous empirical studies.

In Section 2, we give an overview of the semantic interference detection algorithm. The context for this study is discussed in Section 3. Section 4 presents the experimental design and its results. We discuss validity issues in Section 5, and compare our work to related research in Section 6. Finally, we summarize our study and propose future work in Section 7.

2. Overview of Semantic Interference

Our algorithm [21][22] combines data dependency analysis and program slicing. With the data dependency analysis, we can learn about the semantic structure of the program. And the program slicing can identify which semantic structures are impacted by changes. By comparing the overlap of the impacted parts of the two versions, we can learn if they are in conflict.

2.1. Semantic analysis of change impact

Semantic analysis of change impact is the basis for the semantic interference detection algorithm. Semantic program analysis discloses internal dependencies within programs. Although there are many aspects to program semantic analysis, this algorithm only focuses on the local data flow dependencies related to variable def-use pairs. Because we are only concerned with the change impact on local behavior, the semantic analysis is at the statement level.

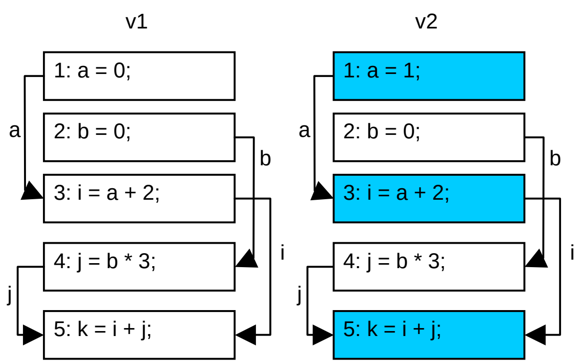

Figure 1 Changes in direct variable def-use dependencies

Figure 1 illustrates the semantic analysis of change impact. Suppose there is a change from version v1 to v2. At first, we will analyze the semantic dependencies in the two versions. In v1, Line 3 uses the value of variable *a* defined in Line 1. So, there is a dependence between Line 1 and Line 3 according to variable *a*. For the same reason, Line 4 has a dependency on Line 2 according to variable *b*, Line 5 has a dependency on Line 3 according to variable *i*, and Line 5 has another dependency on line 4 according to variable *j*. In Figure 1, the variable def-use dependency is represented as solid arrow and the variable names are on the lines.

We use a triple (*var*: *def*, *use*) to represent a dependency, 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. So the dependences in version v1 are {(a: 1, 3), (b: 2, 4), (i: 3, 5), (j: 4, 5)}.

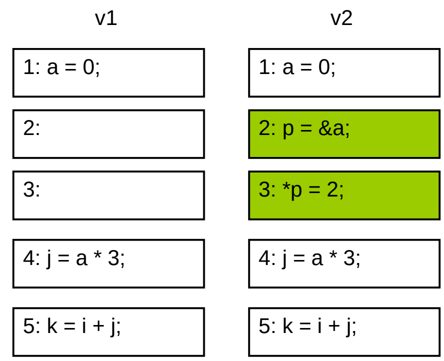

Figure 2 Changes in indirect variable def-use dependencies

From the variable use-def dependency analysis of the two versions, we calculate the change impact by forward slicing from the changed statements. In this example, the change from v1 to v2 modified Line 1 from “a = 0” to “a = 1”. According to the variable def-use chains, {(a: 1, 3), (i: 3, 5)}, we learn that Line 3 and 5 will be impacted. So the impact of this change is Impact (v1->v2) = {3, 5}.

Not all the variable def-use dependencies are easily identified as the “direct” dependencies shown above. There many other indirect dependences that are based on structure, pointer (in C/C++) or reference (in Java and C#) type variables. In Figure 2, adding Line 2 and 3 in Version v2 introduced an implicit dependency on Line 3 and Line 4 based on variable a’s aliasing, *p. The semantic change impact analysis will become more complex and difficult when branches are involved. In this study, we evaluate only direct dependencies and , hence, detect only “direct” semantic interference.

2.2. Semantic Interference Detection

We give a brief introduction to the semantic interference detection algorithm; a detailed explanation can be found in [21] and [22].

This algorithm combines static program slicing and data flow analysis to detect semantic interference. Given two changes to be checked,

1) Calculate the data dependence graph and collect the variable def-use pairs for each version in the changes;
2) Depending on the textual difference between the two versions in a change, identify variable def-use pairs affected by the change;

3) With the affected variable def-use pairs as slicing criterion, do forward slicing to get the program fragments impacted by the change;
4) Compare the impacted fragments of the two changes. The overlapping parts are their interference fragments.

Figure 3 illustrates the semantic interference detection algorithm. Suppose there are two adjacent changes: d1 and d2 where d1 changed the program from version v1 to version v2, while d2 changed the

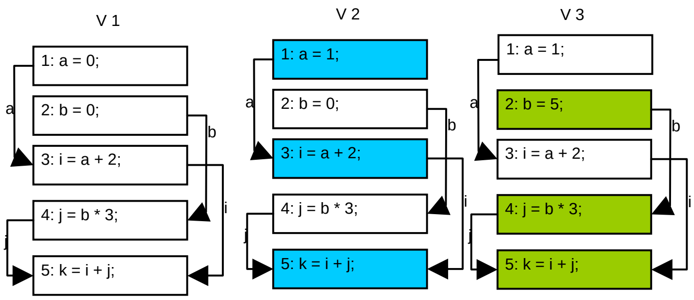

Figure 3 Semantic interference detection procedure.

program from v2 to v3.

1) For each version, calculate data dependence graph and identify variable def-use pair. The results are: for v1, the dependency in is {(a: 1, 3), (b: 2, 4), (i: 3, 5), (j: 4, 5)}; version v2 is {(a: 1, 3), (b: 2, 4), (i: 3, 5), (j: 4, 5)}, and version v3 is {(a: 1, 3), (b: 2, 4), (i: 3, 5), (j: 4, 5)};
2) For each change, identify the changed lines. In d1, Line 1 was changed and in d2, Line 2 was changed,
3) Calculate the semantic impact of the two changes by forward slicing from the changed lines. So, Impact (d1) = {3, 5} and Impact (d2) = {4, 5};
4) Compare impacted lines of the two changes. In this example, Impact (d1) = {3, 5} and Impact (d2) = {4, 5}. The two impact sets overlapped on Line 5. This means Change d1 and d2 have a semantic interference at Line 5.

3. Study context

In this study, the data repository from our previous study constitute the base environment in which to evaluate the semantic conflict analyzer (SCA) and its interference detection algorithm. We present the description of them in this section.

3.1. Change & Version Mgmt Repositories

This study is based on one subsystem of 5ESS™, a successful industrial project with high degrees of parallel changes. 5ESS is a telephone switch project developed by Lucent Technologies [8] and has about 100,000,000 lines of C and C++ code and another 100,000,000 lines in header files and makefiles. Its project structure (feature development) contributed to the high degree of parallel changes during the development process. In this subsystem, the number of developers reached 200 at its peak and dropped to a low of 50. Two products, one for US and one for international customers, were developed separately although some files are common for both of them.

The version and fault history data for our study comes from the change management system of 5ESS™. In Lucent Technologies, the evolution of 5ESS™ is managed by a two-layered system: a change management layer, ECMS [23], to initiate and track changes to the product, and a configuration management layer, SCCS [16], to manage the versions of files needed to construct the appropriate configurations of the product. In 5ESS™, the changes are recorded in a layered hierarchy: Feature, Initial Modification Request (IMR), Modification Request (MR) and delta. A feature is the fundamental unit of extension to the system, and each feature is composed of a set of IMRs that represent problems to be solved. All changes are handled by ECMS and are initiated using an IMR, which may have one or more MRs (each of which represents a solution, or part of a solution, to an IMR's problem), whether the change is for fixing a fault, perfecting or improving some aspect of the system, or adding new features to the system. Each functionally distinct set of changes to the code made by a developer is recorded as a MR by ECMS. For each MR, there is a short abstract written by developers describe its purpose. We use the approach in [9] to classify MRs into according to their purposes: adaptive, perfective, and corrective. When a change is made to a file in the context of an MR, SCCS keeps track of the actual lines added, changed, or deleted. This set of changes is known as a *delta*. For each delta, ECMS records its date, the developer who made it, and the MR to which it belongs. So, from ECMS and SCCS, we can get both the actual changes on the source code and the purpose for the changes.

SCCS is a pessimistic version control system. At a given time only one developer can check out and modify a program. Changes representing different MRs are often interleaved with each other, providing a sequential set of changes but which represent logically parallel changes. *We extend our definition of logically parallel changes further to include those changes made independently and committed by different developers within a short time interval.*

3.2. Parallel changes in this repository

We chose the 5ESS™ subsystem to evaluate our semantic conflict analyzer (SCA) to provide continuity with out previous studies [12] [13], where we found the following:

- There are multiple levels of parallel development. Each day, there is ongoing work on multiple MRs by different developers solving different IMRs belonging to different features within different releases of two similar products aimed at distinct markets.
- The activities within each of these levels cut across common files. 12.5% of all deltas are made by different developers to the same files within a day of each other and some of these deltas interfere with each other.
- Over the interval of a particular release, the number of files changed by multiple MRs is 60% that are concurrent with respect to that release. These parallel MRs may result in interfering changes – though we would expect the degree of awareness of the implications of these changes to be higher than those made within one day of each other.

Furthermore, our previous study also found that there is a significant correlation between files with a high degree of parallel development and the number of faults. Using PCmax, the maximum number of parallel MRs per file in a day, as the measure of the degree of parallel changes, our analysis showed that high degrees of parallel changes tend to have more faults. The analysis of variance strongly indicates that, even accounting for the faults correlated with lifetime, size and numbers of deltas, parallel changes were a significant factor ($p < .0001$).

In this repository we found high degrees of parallel changes and a direct correlation between parallel changes and faults. We believe that this repository serves well to adequately evaluate the utility and effectiveness of the methods, techniques and tools that detect interference between parallel changes.

The focus in [12] and [13] was on textual conflict. It showed that only 3% of the deltas made within 24 hours by different developers physically overlap another's change. The ineffectiveness of textual conflict detection is one of the major reasons to develop a semantic level interference detection algorithm and conduct empirical studies of its effectiveness using industrial/historical data.

3.3. Implementation issues

In [21] and [22], there are two distinct analyses of semantic interference provided: between adjacent versions and between non-adjacent versions. In this study, we implemented and evaluated the adjacent analysis. The non-adjacent analysis needs an extra assumption: the second change should start from a tested and accepted version. According to our knowledge about the 5ESS™ history, this is difficult to guarantee and may not be feasible in practice. To make our study as sound as possible, we used the adjacent analysis that does not require that assumption.

From an analysis on the 5ESS™ code as well as our personal industrial experience, it is clear that industrial projects make significant use of pointers. Because of this we extended our interference analysis one step further to that of de-referenced variables. We consider this still to be a form of "direct" semantic interference because we are not doing pointer analysis as such but still focusing on def-use pairs to determine interference.

Given that we do not do pointer analysis, there is a problem with soundness with respect to interference detection of de-referenced variables: the possibility of what looks like an interference is not – i.e., it is a false positive. But if the developers make safe and disciplined use of pointer assignment, the likelihood of false-positives is eliminated and the analysis of de-referenced variables is as sound as that of ordinary variables. Given the efficiency of local analysis and the avoidance of pointer analyses, we believe it is a useful trade-off. The results of our study support that claim.

The implementation of the data dependency calculation and program slicing is based on GrammaTech's CodeSurfer [1]. For pointer analysis, we select CodeSurfer's option that distinguishes individual fields in a referenced structure. This will use the most precise pointer analysis Codesurfer offers. The C compiler is Visual C++ 6.0.

The C language used in 5ESS™ does not conform to ANSI C. Among other things, the macro "#feature" is not supported by standard C compilers. Thus we made textual changes to the source so that the program could be compiled by Visual C++. Our preprocessing does not change the semantics of the studied programs.

The study is done on a Pentium III 800MHz PC with 256M RAM and Microsoft Windows 2000.

4. Study and Results

From the observation and implications from the previous study, we proposed 3 hypotheses in this evaluation:

1) H1: semantic interference is more likely in the higher degrees of parallel changes.
2) H2: semantic interference is closely related with faults in the high degrees of parallel changes.
3) H3: semantic interference detection is light-weight and efficient.

We prepared three sets of changes that represent these different degrees of parallelism. We ran the semantic conflict analyzer on each set. We compared the results from the three sets to evaluate the effectiveness of the detection algorithm on different degrees of parallel changes. We also estimated the overhead by considering the execution time consumed in running the analyzer.

Our study has 5 steps. We introduce the results and their analyses according to those steps.

4.1. Versions and the degree of parallelism

In this step, we prepared changes to be studied. To supply changes of differing degrees of parallelism, we constructed three equivalent sets of parallel changes from the change and version histories of 5ESS™:

1) For the control set, we randomly selected versions that have no parallel changes with respect to a particular release – that is, the interval between the versions are so long, greater than 1 month, that they can not be viewed as a parallel changes.
2) For the low degree of parallelism set, we randomly selected versions that are logically parallel with a reasonable interval of time (from 1 week to 1 month). In this case, we claim the developers have sufficient time to understand the implications of the changes made by others.
3) For the high degree of parallelism set, we randomly selected versions that are logically parallel with a very short interval time, less than 1 week. In this case, it is difficult, we claim, for the developers to fully understand the changes made by others in such a short time.

Table 1 shows the three sets we selected. To maximize internal validity, we added the following controls while composing the three sets. We made each set as nearly identical with respect to the distribution of different change purposes (adaptive, corrective, and perfective), average size of changes (in number of changed lines), and average size of the source file (in lines of code, LOC).

Set	Version	Adaptive change	Corrective change	Perfective Change	Avg Size (LOC)	File size (LOC)
High	46	35	5	6	45	1.64K
Low	27	19	4	4	53	1.51K
Control	17	11	3	3	48	1.34K

Table 1 The statistic of the sampled three set of changes. The three sets are similar in the distribution of the change purposes, size of changed code and size of source file.

4.2. Calculate semantic interferences

In each set of parallel versions, we use the semantic interference detection algorithm to calculate the conflicts between changes. We compare the density of interference versions in the three sets. The result is in Table 2.

Set	Versions	Interference versions	Interference density
High	46	19	41.3%
Low	27	8	29.6%
Control	17	2	11.8%

Table 2 the semantic interference detected in three sets. In the sampled three set of changes, the higher the parallelism is, the higher the density of semantic interference is.

The high interference density in high degree parallel change sets supports Hypotheses 1: semantic interference is more likely where there are high degrees of parallelism. This result is congruent with our earlier findings in with respect to the degree of concurrency and the likelihood of faults [12] [13].

This result also demonstrates support for our belief that more conflicts happened at semantic level than at textual textural level. In the previous study, in high degree parallel changes, only 3% interference can be detected in the textual level. And the results in Table 2 show that, compared with textual interference, semantic level analysis can disclose significantly more interference than the textual analysis.

4.3. Identifying the relevant faults

Semantic interference by itself does not indicate a fault. The interference might well be intended to correct a fault or to add new processing, etc. It is unintended interference that is likely to represent a fault. What we do then is use semantic interference as a predictor of faults. Some will be faults, some will not. The critical part of this experimental evaluation is to determine on the basis of a fault set for each changed version which of the interferences represent a fault and which do not.

Set	Versions	Interference	Hit	False positive
High	46	19	8	10 (57.9%)
Low	27	8	2	6 (75.0%)
Control	17	2	0	2 (100.0%)

Table 3 The match between semantic interference fragments and faulty code fragments in the three sets of changes. In the three sampled sets, the higher the parallelism is, the lower the false positive is.

To do this, we need to identify the faults following the interference changes. We use the code fragments that are changed in Corrective MRs to represent faults. For each set of parallel versions (each with its set of semantic interferences), we first mine the change management history to look for Corrective MRs subsequent to and dependent on these versions with interference. Then we mine the version management system to get the changed code fragments in these Corrective MRs. We note that regression analysis of our data indicates the results are significant ($p < .001$).

4.4. Evaluating analyzer effectiveness

The effectiveness of the detection algorithm is based on the match between semantic interference and faulty code. We checked the accuracy of the predictions by checking the defect MRs written against those versions with interference. We also studied the semantic properties of our evaluation results. We divide the evaluation into two sections: 1) where SCA detects semantic interference and there is a fault, and 2) SCA detects interference and there is no fault, or where there is a fault but no detected interference.

4.4.1. Match: semantic interference & faults

For each of the three sets, we compare the semantic interference fragments obtained in Step 2 with the faulty code fragments determined in Step 3. We classify the results into 3 groups:

- Hit - detected interferences found in faulty code fragments;
- False positive – detected interferences that have no faulty code fragments associated with them;
- Miss – faulty code fragments that have no detected interferences associated with them.

Table 3 shows the match between the interference and faults. So, if we use the interference detection algorithm to predict faults, the hit rate in high degree parallel change set is much higher than that in the low degree set.

Set	Versions	Fault-related	Fault-related density
High	46	25	54.3%
Low	27	16	59.3%
Contro l	17	10	58.8%

Table 4 Density of fault-related versions in the three sets of changes are very similar.

To check the soundness of the comparison, we also studied the density of fault-related changes in each set. The fault-related changes are deltas that have dependencies [14] with the following corrective MRs, that is, the code fragments changed in these deltas overlapped with the code fragments changed in the later corrective MRs. Table 4 shows that, for the three sets, the distribution of fault related versions are very similar.

4.4.2. Analysis: false positives and false negatives

Besides the above comparisons, we also analyzed the semantic properties of the 3 groups according to the classification of errors found in [20].

1) In *hit* group where the detected interference is found in the faulty code, all the 10 matched interference are non-pointer variable faults. For example,

```
<  i = pos_no;
>  i = pos_no++;
```

In a more specific error classification, all of them are *incorrect variable used* faults. Eight of the 10 faults are *path selection* faults. This means an error in variable usages represents a fault in the computation where the program selected a wrong path. Two of the 10 are *computation* faults, which mean the incorrect variable usage generates erroneous outputs.

2) In *false positive* group, the semantic interferences are detected, but they do are not found in any faulty code. Because the noise level is very critical for a static analysis tool like SCA, we give a further classification on the 19 false positives:

a) Eight of the 19 are *variable or type renaming* cases. For example,

```
< quote_ptr->osps_aq.acronym[i]
=msg_ptr->
text.my_mgacqs.htl_acrnym[i];

> quote_ptr->tsps_aq.acronym[i] =
msg_ptr->
text.my_mgacqs.htl_acrnym[i];
```

b) Five of the 19 are *fault-fixing* cases

This kind of semantic interference is intentionally introduced to fix a fault.

c) Three of the 19 are *false identification of changes*. For example,

```
< (CRoaddrtbl[cid.crindx]->ospsff &
0xfffffffe) | ((DMUNLONG) 1);

>( CRoaddrtbl[cid.crindx]->ospsff &
0xfffffffe) |

>          ((DMUNLONG)1);
```

This kind false positive is an artifact of CodeSurfer and comes from the incorrect identification of same vertex in two versions. In this detection algorithm, the corresponding vertex in the two versions is identified by its type and the associated text. So, in the change above, we view the two statements as different, although only space characters were added in the second version. We did not use semantic equivalence evaluation as found in [24] because it is too expensive to compute in a real project.

3) In the *miss* group, where the faults are *invisible* to our interference detection algorithm, we classify the 17 false negatives according to their semantic properties. This can guide users to utilize our interference detection approach in effective ways.

a) Eight of the 17 are *control flow* faults. For example,

```
<  if ( i < 5)
>  if ( i <= 5)
```

For such kind of change, no impact happened on variable definition and use. So our interference detection algorithm cannot identify such faults that are related to control flow errors.

b) Five of the 17 are faults related to *pointer variables*.

We reviewed the sampled versions and found some pointer arithmetic operations changed the target objects of pointers. But such changes are ignored in our algorithm because Codesurfer assumes that pointer arithmetic will not change the pointer-to set.

c) Four of the 17 are basically *other* faults.

In summary, the match between semantic interferences and predicted faulty code gives support to hypothesis H2. Moreover, the semantic analysis on false positives also confirms that SCA suffers the same problems as other static analysis tools. And for the non-pointer variables that static analysis can give a more precise analysis that for the pointers, SCA can give a more precice prediction of potential faults.

4.5. Evaluating the analyzer efficiency

While evaluating the efficiency of SCA, we use the time used in calculating semantic interference as the overhead. In analyzing the semantic interference on the sample versions, the average time is about 2 minutes. In this overhead, 83% is spent on the program dependency analysis with CodeSurfer, and the time for the calculating and detecting interferences is even smaller. And the average time to compile the analyzed version is 1.8 minutes. This result supports Hypothesis H3: semantic interference is a lightweight and efficient fault prediction approach.

Based on these results, we can claim that the use of SCA is both effective and efficient and will aid developers in finding and fixing faults. As soon as a developer finishes a new version, SCA can give him/her instant feedback about conflicts. The immediate warnings can remind him/her to check or fix faults in the new version while he/she is still familiar with what changes have been made in this new version as well as the reasons for making them. For the versions we studied, it is usually 150 days on average after the fault-inducing changes were made that they were found by various means such as testing, etc. The use of SCA will save the effort of having to rediscover or recall what was done 5 months early.

5. Validity analysis

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

5.1. Construct validity

We focus on a specific and well-defined form of semantic interference between versions. Given that there are multiple levels of parallelism in a large-scale development, we feel that our distinction between high, low and no degrees of parallels is justified for this study evaluating the efficiency and effectiveness of our analyzer. We also claim the delay time construct is well defined and justified as well: the time from the version commit until the opening of the fault MR. The problematic time construct is the fix time. First, one should view that as a maximum possible time where only a portion of that time is actually spent finding and fixing the fault. Second, only a portion of the actual time is spent finding the problem; it is this time that would be saved by our analyzer.

5.2. Internal validity

In evaluating the effectiveness and efficiency of the interference detection algorithm, we used the comparisons among sets of different degree of parallelism. To filter out factors other than parallelism, we have checked the similarity among the sets in distribution of changes of different purpose, the file size, the change size, and the density of fault-related changes. We argue that the equivalence of these factors rules out confounding variables.

We believe that our results our consistent with what one would intuitively expect about parallel changes and what is supported with out earlier studies: highly parallel changes do not allow time for developers to adequately understand the implications of changes and hence are more prone to faults as a result of their changes. Our current results are consistent with our earlier results: there is a significant correlation ($p < .001$) between the degree of parallelism, interferences and faults. The new and interesting results here also agree with ones intuition about adaptive changes: there are likely to be more interfering changes made to add new functionality than in correcting faults, or improving existing functionality.

5.3. External validity

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 system, 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 semantic interference detection algorithm. The only variation in the evaluation process is the use of non-adjacent versions of the interference detection algorithm. This is because, in CVS, the sequential order introduced by the check-in time in SCCS will not be valid.

A threat to our study is that 5ESS™ is a very large-scale real-time project with a large number of developers, geographically distributed. We argue, however, that the subsystem we studied is perhaps by itself more representative of a typical large project. The critical factor, however, is the issue of parallel changes to the same files by different people – i.e., feature ownership rather than code ownership. This form of development is becoming more prevalent, and this supports our claim for external validity.

6. Related work

In this section, we discuss the work related to our semantic interference detection algorithm and the empirical evaluation on software tools with history repositories.

6.1. Program Slicing

Program slicing is an important technique in analyzing properties of a program, where on the basis of some interesting points, or criteria, it computes the affected program entities. In [6], the authors proposed the combination of a program dependency graph and program slicing for conflict detection. Although we use program slicing for semantic interference detection, the different purposes between our work and theirs induced a significant difference in way slicing is used.

The goal of their detection research is for merging, so they check if the changes made in two versions are to be kept in the merge version. Our goal is to check if the later version semantically impacts the earlier one. So we do not need to compose a “merged” version before checking for interference. Further in our algorithm, only one pair of deltas is needed to check interference because the order of the versions breaks the symmetry between the two versions.

In the semantic analysis, we only focus on variable def-uses rather than the whole dependency graph. While such a simplification means we do not catch all possible faults, it does make SCA a lightweight tool that is both feasible and effective in real projects. The results of Step 5, the efficiency evaluation of the detection algorithm, gives strong support for our simplification.

Yang [24] proposed semantic preserving transformations that increase the soundness of semantic conflict detection. Given two versions, this approach can detect the vertices that are equivalent semantically but different textually. This approach can discover some of the false positives that we miss. But the computation to identify vertices with equivalent execution behavior is expensive and would degrade the efficiency of SCA in real applications. Thus, in SCA we identify vertices textually.

6.2. Change impact analysis

[15], [17], and [19] propose change impact analysis based on atomic change classifications, and associate them with test cases. In our detection algorithm, we also check for the effect of changes but differ in three ways:

1) *Category*: They combine static and dynamic analysis and we focus on static analysis.
2) *Granularity*: They work at the method level, and compare two abstract syntax trees providing more precision but paying a higher overhead. We work at the statement level, comparing the vertices by variable def-uses and associated text. Ours is more narrowly focused with significantly less overhead.
3) *Goal*: They build affect-relationships between test cases and atomic changes. This facilitates defect localization by minimizing the related test cases. We focus on lightweight static analysis that can detect interference between versions and predict possible defects.

6.3. Fault-inducing change localization

In our study, the matching between interference fragments and faults has some similarity with the fault-inducing change localization approaches such as [18] where they identify changes from the version management system and a fault database, and correlate them as fault-inducing changes. We similarly identify changes, but differ in how we identify fault-inducing changes. Our approach is based on the semantic analysis on the changes and their interference, while [18] focuses on mining version histories and fault databases.

[25] uses passed and failed test cases to filter out non-related test cases and to select possible fault-inducing changes. Their work focuses on the localization of fault-inducing changes by running test cases, while ours focuses on the prediction of faults with static analysis of changes that semantically interfere.

Program chopping [5] is used in debugging to minimize possible fault-inducing code fragments. Compared with static program slicing we use, dynamic slicing can improve the precision for pointer analysis and reduce false positives in semantic interference detection. But compiling programs and running test cases are required beforehand. From our experience with 5ESS™ and its highly parallel software development, these preparation steps will take a significant amount of time. How to effectively incorporate such dynamic approaches to improve the soundness and effectiveness of our approach is a possible avenue of future work.

6.4. Empirical uses of repositories

There are an increasing number of empirical studies using version control repositories. The information from version management systems can provide useful information from the real projects, non-intrusively, such as fine-grained source code changes, complete histories of all the changes, etc.

Atkins et al. [2] uses change history and an effort estimation model [4] to calculate the effort that has been saved using the version editor (VE). This study illustrates an important point: the versions found in the repository were separated into two groups based on a criterion that was useful in the empirical study and a significant result was obtained on the basis of this differentiation. The quantitative results provide evidence showing the usefulness of the software tool.

But for our static semantic interference analysis approach, the number of false positives is a primary concern for real applications. Thus, as an evaluative study, we mainly focus on the effectiveness, and only use time to estimate its efficiency. As we have positive results in this step, we could use the effort estimation model in [4] to quantify the saved effort using our tool.

However, the origin of the data in our study is quite different from theirs. In the construction of the control and treatment groups, they were able to differentiate the historical data into VE and non-VE related groups. Without that extra-repository distinction (i.e., the VE footprint instrumented into the histories), the evaluation would not have been possible. Thus, while this empirical study did use historical data from version and change management, it was, in a real sense, instrumented data. While in our evaluation of SCA, all the data in our study was not instrumented, but data readily available in the repositories.

Compared with [3], we are similar in mining changes history to predict faults. But their granularity is large: the number of changes on a file, or the number of lines changed in a period of time. They do not consider interference between changes, whether at the textual level or the semantic level.

[26] also searches for association relationships between changes by mining change histories to predict possible changes in the future. But their granularity is also different from ours. They focus on structure-related entities, such as fields or functions in a file, or files in a directory, and predict faults from the incomplete changes. We, on the other hand, focus on the semantics of the code, predicting faults from semantic interference.

7. Summary and Conclusions

Our research has yielded two important contributions: first, we have shown that a limited form of semantic interference detection can provide an effective and useful means of predicting faults in an increasingly common context; and second, we created an effective and novel design for rigorous experimental evaluation of analysis tools using and mining change and version management repositories.

While our technique is incomplete both in detecting semantic interference and in predicting faults, it is both efficient (taking roughly the time of a compilation) and effective in predicting faults. There is a problem with soundness in detecting interference caused by de-referenced variables but if the developers practice safe and disciplined use of pointers and references (i.e., no pointer assignments between the definition and use of de-referenced variables), even that analysis would be sound. Despite our tool suffering the same problems as other static analysis tools (the unsoundness due to the imprecision of pointer analysis), the resulting analyses and predictions are still extremely useful as the false positives are easily dealt with by the developers.

The results of our evaluative experiment are as follows:

1) Semantic interference is significantly higher in high group (twice the other two);
2) Semantic interference detection is effective in predicting non-pointer variable faults from the detected interferences in versions changed within short time periods
3) Of the detected semantic interferences, 40% in the high group matched with known faults, 25% in the low group (33% in both). The detected interferences matched 20% of all the faults in all the versions we analyzed.
4) The overhead of using our semantic interference detection approach is very low and can help developers to find faults very early.
5) Preciseness of pointer analysis, identification of variable renaming, and control-flow changes are the major factors that affect the effectiveness in detecting interference and predicting faults.

Our experimental design itself is a significant contribution to providing rigorous evaluation of tools. We avoid the invalidity problems of contrived. Change and version management repositories provided a large enough population to obtain a variety of sample data such as change purpose, size of changed code, and size of source file. With classifying the sampled data according to the degree of parallelism, we constructed a control group and two treatment groups.

To provide an effective evaluation, the fault sets were mined from the version and change management repositories, rather than intentionally introduced. The "mundane realism" not only removes the *internal validity* problems associated with fault seeding (the representativeness of the faults seeded, the placement of those faults, and the frequency of fault occurrence), but also increases the *external validity* or the study.

8. Future Work

The results from our study also suggest ways of combining our approach with others to improve the effectiveness of semantic interference detection and fault prediction:

1) *Increase semantic interference detection completeness*. Our tool is sound in detecting the direct interferences related to ordinary data dependencies, but it misses the interferences related to control flow and pointer analysis that contributed to many of the faults in our study.
2) *Identify false positives in semantic interference detection*. Although it is not very difficult for developers to manually inspect and dismiss false positives, improved and automated identification of intended interference would reduce the noisiness of our analyses. We will look at dynamic analysis techniques, such as dynamic slicing [5] or symbolic execution [7], to identify interferences more precisely.
3) *Reduce the preparator workload for semantic analysis*. In SCA, all the versions must be compiled before data dependency analysis is possible. This is not always the case in real projects. Island grammar [10] [11] may be a light-weight approach to detection without the need of compilation.

9. Acknowledgements

We thank Harvey Siy, University of Nebraska, Omaha, for his help on the change management system of 5ESS™ and his suggestion about island grammars; Barbara Ryder, Rutgers University, for her review of preliminary results; and Xiangyu Zhang, Purdue University, for the discussion on dynamic slicing issues. This work was supported in part by NSF CISE Grant IIS-0438967

10. References

[1] P. Anderson, and T. Teitelbaum, "Software Inspection Using CodeSurfer", *Proc. of the First Workshop on Inspection in Software Engineering (WISE'01)*, Paris, France, July 2001, pp. 4-11.

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus. "Using version control data to evaluate the impact of software tools: A case study of the version editor", *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, July 2002, pp. 625–637.

[3] 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. pp 653-661.

[4] T.L. Graves, A. Mockus, "Inferring Change Effort from Configuration Management Databases", *Pro. of the Fifth International Symposium on Software Metrics, IEEE*, 1998, pp. 267-273.

[5] 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, November 2005, pp. 263-272.

[6] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, July 1989, pp. 345-387.

[7] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing", *Proc. of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, Warsaw, Poland, Apr 2003, pp. 553-568.

[8] K. Martersteck, and A. Spencer, "Introduction to the 5ESS™ Switching System", *AT&T Technical Journal*, Vol. 64, No. 6, part 2, July-August 1985, pp. 1305-1314.

[9] 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, pp. 120-130.

[10] L. Moonen, "Generating Robust Parsers Using Island Grammars", *Proc. of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Stuttgart, Germany, October. 2001, pp. 13-22.

[11] L. Moonen, "Lightweight Impact Analysis Using Island Grammars", *Proc. of Tenth International Workshop On Program Comprehension (IWPC'02)*, June 2002, pp. 219-228.

[12] D.E. Perry, and H.P. Siy, "Challenges in Evolving a Large Scale Software Product", *Proc. of the International Workshop on Principles of Software Evolution, the 20th International Software Engineering Conference*, Kyoto, Japan, April 1998, pp. 251-260.

[13] 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, pp 308-337.

[14] R. Purushothaman, and D.E Perry, "Towards 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, pp. 511-526.

[15] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs", *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications(OOPSLA 2004)*, October 2004, pp. 432-448.

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

[17] B.G. Ryder, and F. Tip, "Change impact analysis for object-oriented programs", *Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, June 2001, Snowbird, Utah, pp .46-53.

[18] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? On Fridays", *Proc. of International Workshop on Mining Software Repositories (MSR)*, Saint Louis, Missouri, May 2005.

[19] M. Stoerzer, B.G. Ryder, X. Ren, and F. Tip, "Finding Failure-Inducing Changes using Change Classification", *Research Report RC 23729*, IBM, September 2005.

[20] K. Tewary and M.J. Harrold, "Fault Modeling using the Program Dependence Graph", *International Symposium on Software Reliability Engineering*, November 1994, pp. 126-135.

[21] G.L. Thione, "Detecting Semantic Conflicts in Parallel Changes", MSEE Thesis, The Department of Electrical and Computer Engineering, The University of Texas at Austin, December 2002. 98pp.

[22] 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, pp. 47-56.

[23] P.A. Tuscany, "Software development environment for large switching projects", *Proc. of Software Engineering for Telecommunications Switching Systems Conference*, 1987.

[24] 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, pp. 310-354.

[25] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" *Proc. of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, Vol. 1687 of LNCS, Toulouse, France, September 1999, pp. 253-267.

[26] T. Zimmermann, P. Weibgerber, S. Diehl, A. Zeller, "Mining Version Histories to Guide Software Changes", *Proc. of the 26th International Conference on Software Engineering (ICSE 2004)*, Edinburgh, UK, May 2004, pp. 563-572