Mining Change and Version Management Histories to Evaluate an Analysis Tool
– Extended Abstract –

Danhua Shao, Sarfraz Khurshid and Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
Electrical and ComputerEengineering,
The University of Texas at Austin, Austin TX 78712

{dshao, khurshid, perry}@ece.utexas.edu

ABSTRACT

Parallel changes are becoming increasingly prevalent in the development of large scale software system. To deepen the study on the relationship between parallel changes and faults, we have designed a tool to detect the direct semantic interference between parallel changes. In this paper, we describe an empirical study to evaluate this semantic interference detection tool. We first mine the change and version management repositories to find sample versions sets of different degree of parallel changes. On the basis of the analysis reports, we mine the change and version repositories to find out what faults were discovered subsequent to the analyzed versions. We will also determine the lapse and cost data for finding and fixing the faults associated with those samples. This approach provides a significant and low cost method for effectively evaluating the usefulness of software tools.

1. INTRODUCTION

The evaluation of software engineering tools is best served by a rigorous empirical approach. A solid empirical basis serves to provide a deep understanding of the problem of how to determine the effectiveness of tools in software development. So, based on the delineation of the parallel changes in the large scale software systems and the algorithms to detect the potential interference between them, we want to evaluate the effectiveness of this detection method.

This research is part of a continuing series of investigations based on the change and version management histories of one subsystem of Lucent Technologies 5ESS Telephone Switching System. Upon the analysis of the change history, we found that high degrees of parallel changes happened during the development [1, 2]. So these changes are very likely to conflict with each other.

We classify the conflicts into two levels: *prima facie* conflicts (the changes happen on the same lines of source code) and *semantic* conflicts (the changes are made to the same slice of program and modify the semantics in different ways). In a previous study on the subsystem of 5ESS, we showed that 3% of the changes made with in 24 hours by different developers physically overlapped each others' change [1, 2].

But we believe that more conflicts exist at the semantic level. Based on the data dependency analysis and program slicing, we have proposed a semantic interference detection algorithm [15].

To investigate the effectiveness of this algorithm, we propose to evaluate the following two hypotheses: 1) the number of semantic interferences this algorithm will detect is dependant on the degree and kind of parallelism that occurs in the versions analyzed; and 2) this algorithm saves time and effort especially where parallel changes happen in the software development. To quantify the evaluation, we use the change and version management histories of the 5ESS subsystem. We also utilize the effort analysis technique in [18] to investigate the effectiveness of our algorithm.

In Section 2, we discuss the background for our evaluation work. In Section 3, we give an overview of the semantic interference detection algorithm. While in Section 4, we show the concrete steps we perform for the evaluation. Finally, in Section 5 we summarize our approach.

2. BACKGROUND

In this study, the data repository and our previous study constitute the base environment to evaluate the semantic

interference detection algorithm. We present the description about them in this section.

2.1 Change & Version Management Repositories

The data for us to evaluate the analysis tool, as well as that for our previous parallel changes studies, comes from the complete change and version management history of a subsystem of the Lucent Technologies' 5ESS. It covers the first fifteen years of the subsystem project [3, 4]. Changes in the development organization also contributed to the high degree of parallel changes in the development. The number of developers reached 200 at the peak and dropped to a low of 50. And the two products, one for US customers and one for international customers, were developed separately although some files are common for both of them.

Lucent Technologies uses a two-layered system for managing the evolution of 5ESS: a change management layer, ECMS [4], to initiate and track changes to the product, and a configuration management layer, SCCS [5], to manage the versions of files needed to construct the appropriate configurations of the product.

In 5ESS, the changes are managed 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 to part of the 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 the ECMS. 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, the ECMS records its date, the developer who made it, and the MR to which it belongs.

2.2 Parallel changes in the repository

Based on this package of data, we have done research on the phenomena of parallel changes and their effects on software quality.

On the parallel change, our investigation showed that [1, 2], in this repository:

- 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.

Further more, our study also found that there is a significant correlation between files with a high degree of parallel development and the number of faults [1, 2]. We use PCmax, the maximum number of parallel MRs per file in a day, as the measure of the degree of parallel changes. The boxplot (Figure 1) from our analysis show that high degrees of parallel changes tend to have more faults. The analysis of variance strongly indicates that, even accounting lifetime, size and numbers of deltas, parallel changes were a significant cause of faults ($p < .0001$).

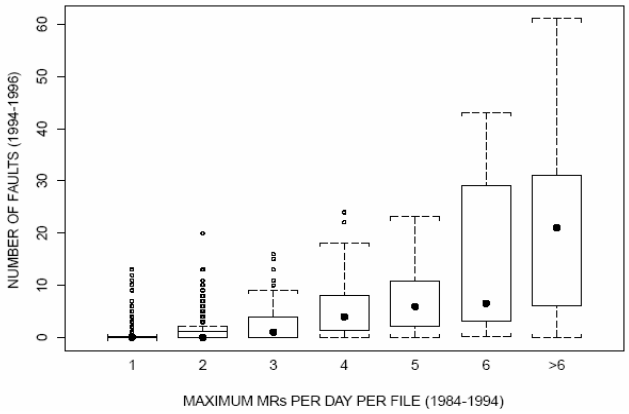

Figure 1: Parallel development (PCmax) vs. number of faults. This boxplot show the number of faults for each file, grouped by degree of parallel changes.

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

3. SCA: A SEMANTIC INTERFERENCE DETECTOR FOR PARALLEL VERSIONS

In the previous study about the relationship between parallel changes and faults, our initial focus was on *Prima facie* conflict. It showed that 3% of the deltas made within 24 hours by different developers physically overlap another's change. But the physical overlap is just one way by which one developer's changes could interfere with another's. We believe that many more interfering conflicts arise as a result of parallel changes to the same data flow or program slice - that is, conflicts arise as a result of semantic interference in addition to syntactic interference. Semantic interference has become one of the major focuses of our current research on parallel changes.

In the study on semantic interference, parallel changes are classified into two categories: temporal and logical.

- Temporally parallel changes are changes performed through Version Management Systems that allow multiple valid copies of the same code module at the same time. Changes of this sort are truly parallel and in general, there is no assumed priority among changes. The final version is an integrated version. These systems embody what is referred to as *optimistic version control* [2, 7]. In this case, there is only one delta per version.
- Logically parallel changes are changes made independently on the same release and may be temporally separated by minutes, days or even months. In this case, the time of submission for a change determines its placement in a priority based ordering. These changes are done in the context of configuration management systems that embody *pessimistic version control* [2, 7]. In this case, there may be several deltas per logical version.

The algorithm to detect semantic interference between two deltas is the kernel component in SCA. We use a combination of the (static) program slicing [11] and data flow analysis [12] (specifically the flow of values from a source node to their destination nodes) as the basis for this analysis algorithm.

For each delta, two dependency sets $\Delta 1$ and $\Delta 2$ are calculated from the version before and after the change. Where each dependency set is a triple (v, d, u) such that vertex u uses the variable v that is defined in vertex d. By comparing the two dependency sets, we can tell from the vertexes which dependency relationships are changed by that delta. Then, through forward slicing, we can get the program fragments impacted by this delta. With the fragment set for each delta, we know whether the two deltas semantically interference with each other.

The implementation of SCA is based on standard real-world development components, such as SCCS for the version management system, and GrammaTech's CodeSurfer [12] for the slicing mechanism. Note that with SCCS, we have only logically parallel versions.

4. MINING CHANGE AND VERSION MANAGEMENT HISTORIES

The design and implementation of SCA, together with the change and version history databases from 5ESS subsystem, provides the experimental setting to evaluate the semantic interference detection algorithm. From the observation and implications from the previous study, we will test the two hypotheses about the tool evaluation:

1. The number of semantic interferences detected by SCA is dependant on the degree and kind of parallelism that occurs in the versions analyzed.
2. SCA saves the time and effort especially where intertwined logically parallel changes occur.

We evaluate the effectiveness of SCA in the following 5 steps.

Step 1. Create three sets of program versions for analysis by SCA

To supply changes of differing degrees and kinds of parallelism, we will construct three sets of logically parallel change versions from the change and version histories. To maximize internal validity considerations, we add the following controls: we make each set as nearly identical with respect to the size of changes, number of faults, and the purposes (i.e., corrective, perfective or adaptive classified according to the MR classification method by Mockus and Votta [20]) of the changes. Further we will choose the number of samples for each set to achieve a significance level of at least .01.

1) For the control set, we randomly select versions that have no parallel changes with respect to a particular release – that is, versions that have only one set of changes for the entire release.

2) For the set representing a low degree of parallelism, we randomly select versions that are logically parallel with respect a specific release, but that have a reasonable amount of time (measured in terms of days or weeks) separating the sequence of versions – sufficient time for the developers to be able to understand the implications of those changes. We choose a representative number of versions with two sets of changes, three sets of changes, etc.

3) For the set representing a high degree of parallelism, we randomly select versions that are logically parallel with respect to a specific release, but represent versions covered by MRs that are in fact being worked on

concurrently (such that the there is a significant degree of interleaving of deltas for each of the files begin changed under each MR – cf. [1,2]). We again choose versions that represent the varying degrees of parallelism from 2 to 16 (the highest degree of parallelism found in [1,2]).

For each set, we mine the change management history to find the candidate versions that match the attributes desired for the overall controls needed and for the specific attributes appropriate for each set. Once we have the candidate set of versions, we then mine the version management history for those versions to be analyzed.

Step 2. Analyze the three sets of parallel changes.

In each set of parallel versions, for each related pair of versions, create the delta pairs needed by SCA to perform its analysis. Each analysis results in a set of semantic interferences created by the second version changing the first. These interferences are the basis for evaluating the effectiveness of SCA.

Step 3. Determine the history of relevant faults.

For each set of parallel versions (each with its set of semantic interferences) we first mine the change management history subsequent to these versions for faults found in these versions and then mine the version management history to get the code fragments changed as a result of fixing those faults. The results of mining these two repositories provide us with the historical evidence of faults relative to the changes in our three version sets.

Step 4. Evaluate the effectiveness of SCA.

The goal of the evaluation step is to determine the time and effort saved in finding the faults at the time of version deposit by determining the time lapse between the time of the fault insertion and its detection and the amount of effort needed to fix the problem when it was detected. With the results from Step 2 and 3, we have the basic data to perform this evaluation.

For each related pair of versions in the three version sets, we compare the program fragments of the faults found and the sets of interference between them, and classify them into three groups:

1. Match - the detected interference represents a fault;

2. No match – an interference was found, but no corresponding fault was found;

3. No match – a fault was found, but no there was no detected interference that matched it.

For group 1, we will analyze and evaluate the performance of SCA in two ways:

a. Determine the time saved by SCA. For set 1,we mine the change management history for the available data on the time the fault was found and the time it was fixed, etc, to calculate the time saved by SCA.

b. Determine the effort saved by SCA. With the information about the changes, we can use the effort estimation algorithm in [18] to quantify the saved effort.

We then analyze the time data across the three different version sets, the fault rates, the change sizes and their respective purposes to provide the time effectiveness of SCA with respect to faults found.

Group 2, represents, presumably, false positives in our interference analysis – that is, interferences for which there were no faults. This represents the noise level in the evaluation of SCA, which if too high will of course detract from its utility as an effective analysis tool. We analyze what appears to be the noise level in two ways:

a. For the false positives that we determine to be intentional interferences, we first check to determine the relationship between corrective changes and false positives. In particular, we want to see how many of the false positives could be eliminated by not analyzing corrective changes. For the remaining false positives, look for common factors among them and consider ways to improve the interference detection algorithm.

b. If the interference is not an intentional one, analyze the code change to see if it represents a fault that has not been found. If it does represent a fault that has not been found yet in the course of system testing and development, this represents an additional dimension of effectiveness for SCA.

Group 3 represents faults that are not visible to SCA in terms of semantic interferences as it is currently configured. We summarize the kinds of faults represented to see whether there might be the faults that could be detected by extending SCA.

At this point, we can determine the relationship between the three version sets and how effective SCA was in each of these sets.

Step 5. Summarize our evaluation work.

With the results of Step 4, we evaluate SCA with respect to how much time and effort it saved, what kinds of faults it can find and what kinds faults it cannot find.

With the evaluation in hand, we add both to our understanding of the amount of interference in terms of both *prima facie* and *semantic* interference in parallel changes and to our understanding of the relationship

between the degree of parallelism in changing source code and the number of faults found.

5. SUMMARY AND NEXT STEPS

The high degree of parallel changes in the development of large scale software systems makes the evaluation of semantic interference detection methods increasingly important. With the data from change and version management histories in the 5ESS system, we construct an empirical study to show how much time and effort our algorithm can save for developers. In this paper, we give an overview of our semantic interference detection method, the data used to evaluate them and the steps to implement this study.

To deepen the evaluation, we would like to do further work following this study:

First, deeply analyze the faults not found by the semantic interference detection algorithm. We could classify them according to their semantic traits and explore various analysis technologies.

Second, compare the cost and effectiveness between direct semantic interference detection and indirect detection. In our current algorithm, we focus only on the direct semantic interference. To quantify the comparison, we should implement an indirect semantic detection algorithm, test it with the same version sets, and compare the resource costs, the detected faults, the ratio of faults to noise, and other aspects between the two detection algorithms.

Third, detect higher levels of interference. In this study, we analyze the version change at the source code level. From the multiple levels of parallel changes in the 5ESS system, we believe that the conflicts between changes will also happen in the higher levels of software development, such as software architecture.

6. References

[1] Dewayne E. Perry, and Harvey P. Siy. “Challenges in Evolving a Large Scale Software Product”, Proceedings of the International Workshop on Principles of Software Evolution, 1998. International Software Engineering Conference, Kyoto, Japan, April 1998.

[2] Dewayne E. Perry, Harvey P. Siy, and Lawrence 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.

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

[4] P.A. Tuscany. “Software development environment for large switching projects.”, In Proceedings of Software Engineering for Telecommunications Switching Systems Conference, 1987.

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

[6] Ulf Asklund, and Boris Magnusson. "A Case Study of Configuration Management with ClearCase in an Industrial Environment.", System Configuration Management, 1997.

[7] Tom Mens. "A State-of-the-art Survey on Software Merging.", IEEE Transactions on Software Engineering, Vol. 28, No. 5, May 2002.

[8] Frederick P. Brooks, Jr. "No silver bullet: Essence and accidents of software engineering." IEEE Computer, April 1987, pp 10-19.

[9] Susan Horwitz, Jan Prins, and Thomas Reps. "Integrating Noninterfering Versions of Programs.", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp 345-387.

[10] J. Estublier, and R. Casallas. “The Adele configuration manager.”, In W.F. Tichy, Ed., Configuration Management: Trends in Software, John Wiley & Sons, 1994.

[11] Frank Tip. “A survey of program slicing techniques.”, Journal of Programming Languages 3, 1995, pp 121-195.

[12] P. Anderson, and T. Teitelbaum. "Software Inspection Using CodeSurfer." In Workshop on Inspection in Software Engineering (CAV 2001), Paris, France, July 18-23, 2001.

[13] Ranjith Purushothaman, and Dewayne E. Perry. "Towards Understanding the Rhetoric of Small Changes - Extended Abstract.", International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering 2004 (ICSE 2004), May 2004, Edinburgh, Scotland.

[14] Ranjith Purushothaman, and Dewayne E. Perry. “Towards Understanding the Rhetoric of Small Changes.”, Accepted for publication. TSE – under review 2005.

[15] Giovanni Lorenzo 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.

[16] Giovanni Lorenzo Thione, and Dewayne E. Perry. “A Technique for Detecting Direct Semantic Conflicts in Parallel Changes.”, Revised March 2004. Submitted for publication.

[17] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mokus. “Using Version Control Data to Evaluate the impact of Software Tools: A Case Study of the Version Editor.”, IEEE Transaction on software engineering, Vol. 28, No. 7, July 2002.

[18] Todd L. Graves, and Audris Mockus. “Inferring change effort from configuration management data.”, In Metrics 98: Fifth International Symposium on Software Metrics, Bethesda, Maryland, November 1998, pp 267-273.

[19] Todd L. Graves, and Audris Mockus. “Identifying Productivity Drivers by Modeling Work Units Using Partial Data.”, Technometrics, Vol. 43, No. 2, May 2001, pp 168-179.

[20] Audris Mockus, and Lawrence G. Votta. “Identifying Reasons for Software Changes using Historic Databases.”, Proceedings of the International Conference on Software Maintenance (ICSM'00), 11-14, October 2000, San Jose, California, USA, pp 120-130.