Parallel Changes in Large Scale Software Development: An Observational Case Study

Dewayne E. Perry
University of Texas at Austin
perry@ece.utexas.edu

Harvey P. Siy
Lucent Technologies
hpsiy@lucent.com

Lawrence G. Votta
Motorola, Inc.
lvotta1@email.mot.com

ABSTRACT

An essential characteristic of large scale software development is parallel development by teams of developers. How this parallel development is structured and supported has a profound effect on both the quality and timeliness of the product. We conduct an observational case study in which we collect and analyze the change and configuration management history of a legacy system to delineate the boundaries of, and to understand the nature of, the problems encountered in parallel development. The results of our studies are 1) that the degree of parallelism is very high–higher than considered by tool builders; 2) there are multiple levels of parallelism and the data for some important aspects are uniform and consistent for all levels; 3) the tails of the distributions are long, indicating the tail, rather than the mean, must receive serious attention in providing solutions for these problems; and 4) there is a significant correlation between the degree of parallel work on a given component and the number of quality problems it has. Thus, the results of this study are important both for tool builders and for process and project engineers.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement – *version control*; D.2.8 [**Software Engineering**]: Metrics – *process metrics*; D.2.9 [**Software Engineering**]: Management – *programming teams, software configuration management*; K.6.3 [**Management of Computing and Information Systems**]: Software Management – *software development*

General Terms: Management, Measurement

Additional Key Words and Phrases: Change management, parallel/concurrent changes, parallel versions, merging interfering and non-interfering versions, software integration

1 INTRODUCTION

Large scale software development presents a number of significant problems and challenges to software engineering and software engineering research. In our pursuit of a deep understanding of how complex large scale software systems are built and evolved, we must understand how developers work in parallel. Indeed, in any software project with more than one developer, parallel changes are a basic fact of life. This basic fact is compounded by four essential [2] problems in software development: evolution, scale, multiple dimensions of system organization, and distribution of knowledge.

- Evolution compounds the problems of parallel development because we not only have parallel development within each release, but among releases as well.
- Scale compounds the problems by increasing the degree of parallel development and hence increasing both the interactions and interdependencies among developers.
- Multiple dimensions of system organization[1] [19] compounds the problems by preventing tidy separations of development into independent work units.
- Distribution of knowledge compounds the problem by decreasing the degree of awareness in that dimension of knowledge that is distributed.[2]

Thus, a fundamental and important problem in building and evolving complex large scale software systems is how to manage the phenomena of parallel changes. How do we support the people doing these parallel changes by

[1] By system organization, we mean the hardware and software components which make up the product. It is not to be confused with the developers' organization.

[2] Here there are two possibilities of knowledge centralization: the knowledge of a part of the system, or the knowledge of (part of) the problem to be solved. If one centralizes knowledge of the system (for example, by file ownership where only the file owner makes changes) then one must distribute knowledge of the problems to be solved over the file owners. Conversely, as is done here, if one centralizes knowledge of the problems (for example, by feature ownership) then one must distribute the knowledge of the system over the feature owners.

organizational structures, by project management, by process, and by technology? How can we support this kind of parallel change effort and maintain the desired levels of quality in the affected software? We are particularly interested in the problems of technology and process support.

Before we can adequately answer these questions we need to understand the depth and breadth of the problem and correlate it to the related quality data. To explore the dimensions of this phenomena, we take a look at the history of a subsystem of Lucent Technologies' 5ESS® telephone switch [16] to understand the various aspects of parallel development in the context of a large software development organization and project.

We use an observational case study method to do this empirical investigation. We describe this study as observational since it captures many important quantitative properties associated with the problem of concurrent changes to software. We consider it to be a case study because it is one specific instance of the observed phenomena.

Central to this technique is an extended series of repeated observations to establish credibility [24]. In this way, the method is similar to the ones used in Astronomy and the social sciences [11]. Finally, a theory is built using these observations (e.g., with grounded theory [7]) to make predictions (hypotheses) that are tested with future studies.

Our strategy for understanding the problem of parallel changes is to look at the problem from a number of different angles and viewpoints in the context of a large-scale, real-time system and a large-scale development. We have three goals in this initial study. First, we provide a basic understanding of the parallel change phenomena that provides the context for subsequent studies. For this we provide basic observational data on the nature of parallel changes. Our thesis is that these problems cannot be (and indeed have not been) adequately addressed without quantitative data illustrating their fundamental nature.

Second, we begin an investigation (which we will continue in subsequent studies) of an important subproblem: interfering changes. Given the high degree of parallelism in our study system and the increasing emphasis on shorter development intervals, it is inevitable that some of these changes will be incompatible with each other in terms of their semantic intent. Here we look at the *prima facie* cases where we have changes to changes and changes made within the same day. In subsequent studies we will explore the extent to which parallel changes interfere with each other semantically (that is, they affect the data flow with in the same slice).

Third, we explore the relationship between parallel changes and the related quality data. We have several hypotheses about this relationship. First, interfering changes are more likely to result in quality problems later in the development than non-interfering changes. Second, files with significant degrees of parallel changes are likely candidates for code that "decays" over time. The degree of interference increases this likelihood. Third, technology supporting the management of these problems address only superficial aspects of these problems.

We first summarize the various kinds of tools that are available to support parallel development. We then describe the context of this study: the characteristics of the organizational, process and development environment and the characteristics of the subsystem under investigation. We do this to provide a background against which to consider the phenomena of parallel changes. Having set the context for the study, we present our data and analyses of the parallel change phenomena, the extent and magnitude of interfering changes, correlate the parallel change phenomena to the quality data, and discuss the construct, internal and external validity of our study. Finally, we summarize our findings, evaluate the various means of technological and process support in the light of our results, and suggest areas for further research and development.

2 RELATED WORK

2.1 Configuration Management

Classic configuration management systems in widespread use today, SCCS [20] and RCS [22], embody the traditional library metaphor where source files are checked out for editing and then checked back in [9]. They induce a sequential model of software development. The locking for a *checkout* operation guarantees that only one user can change a particular file at a time and blocks other developers from making changes until a *checkin* operation has been done thereby releasing the lock on the file. There is no checking for the presence of conflicts between successive changes. The purpose of the configuration management system is to guarantee that, like a database, no changes are lost due to race conditions.

2.2 Management of Parallel Changes

One of the standard features of even the classic configuration management systems that enables developers to create parallel versions is the branching mechanism. Everytime a developer needs to create a new version of the code, he requests the configuration management system to create a new branch. The different versions are all stored in the same physical file. The configuration management system can isolate changes made to one

version from those of other versions by examining the branch identifier essociated with each change. However, since all changes are stored in the same file, development is still inherently sequential as only one developer at a time can make changes to the file.

Newer configuration management systems such as Rational's ClearCase® [14] and the Adele Configuration Manager [6] allow developers to work in parallel on the same file without waiting for some other developer to release his lock on the file. ClearCase's views and Adele's workspaces enable developers to create different versions of the file. Within each view or workspace, a developer can make changes to the code in parallel with other developers.

2.3 Integration of Parallel Changes

The creation and maintenance of parallel versions give rise to another set of problems and issues, depending on whether the versions are permanent or temporary. Permanent versions are "branches in the product development path that have their own life cycle" [15]. These typically mean different releases or different members of a software product family. In dealing with permanent versions, problems arise in managing software product families, in sharing and reusing common code, in propagating common changes across different versions, and in identifying the version best suited for a given application. Temporary versions on the other hand are meant to be merged eventually and only need to exist for the time needed until merging. The problem here is in figuring out how to merge the multiple versions back into a coherent single version, resolving potential conflicts that might arise in the process. We narrow the scope of this paper to this problem. We examine previous configuration management research to address this problem. We will also examine related research into two key issues with integrating parallel changes that are not addressed by configuration management systems: semantic conflicts and logical completeness.

In classic configuration management systems, this merging process has to be done manually. There are no mechanisms to collapse branches back together. Modern configuration management systems provide mechanisms for automatically merging several versions back together. For example, ClearCase provides support for merging up to 32 versions. Mutually exclusive changes are merged automatically. Changes that are not mutually exclusive must be resolved manually and the merge tool provides an interface for doing so.

In one project case study [14], ClearCase was able to automatically merge over 90% of the changed files. The rest required manual intervention. In about 1% of the cases, the merge tool inappropriately made an automatic decision, but nearly all of those cases were easily detected because they resulted in compiler errors. This data came from an in-house merge of the Windows port of ClearCase with their UNIX version [13]. The merge involved several thousand files resulting from nine to twelve months of diverging development effort by about 10 people.

Adele provides a mechanism to automatically merge a file in a workspace into the current version of the file. However, it is recommended that frequent merges be performed by the different workspaces because the probability of conflicts rapidly increases with the number of changes performed in all the copies [6]. Thus Adele requires frequent updating of the changes being made in the other workspaces to keep the various parallel versions more or less in synch.

Semantic Conflicts

Configuration management systems are only able to detect the most simple types of conflicting changes: changes made on top of other changes, that is, changes in one version that physically overlap with changes in another version. There may be many more changes that indirectly conflict with each other. To detect these, more sophisticated program analysis techniques are needed, for instance, the work of Horwitz, Prins and Reps on integrating noninterfering versions [10]. They describe the design of a semantics-based tool that automatically integrates noninterfering versions, given the base version and two derived but parallel versions. The work makes use of dependence graphs and program slices to determine if there is interference and, if not, to determine the integration results.

Logical Completeness

In trying to synchronize a consistent build of a system, we have to worry about logical completeness of changes — that is, we have to worry about dependencies that are shared across multiple components in the system [19]. Cusumano and Selby [4] noted this problem in the course of applying Microsoft's *synch and build* strategy to Windows NT. Their solution to coordinating changes [4, page 273] was to post the intent to check in a particular component and for related files to prepare and coordinate their changes so as to be able to synchronize a consistent build.

This problem of coordinating changes is certainly an important one in the context of large scale system builds out of separately evolved components. This exacerbates the problem of increasing parallel changes, especially for central components which may be indirectly affected by multiple logical changes.

2.4 Empirical Evaluation

We have presented some of the past research done to address the problems of parallel changes. In general,

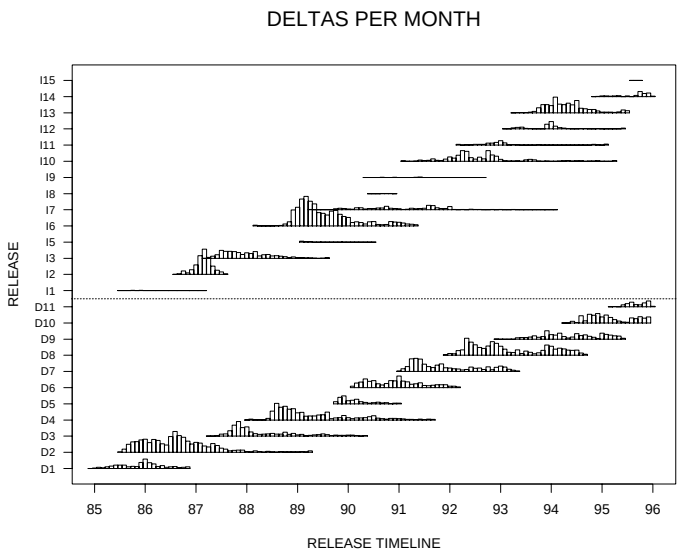

Figure 1: **Timeline of parallel releases.** Each histogram represents number of deltas made per month for one release of the software. The top and bottom halves show releases for the international and domestic products, respectively. In this picture, Release I6 shows a peak of around 1,800 deltas per month.

the studies reviewed here have either been conducted to prove the technical feasibility of specific solutions or to document observations on effective practices of software development. We note a dearth of empirical studies in the literature investigating the scope and problems due to parallel changes.

The data offered in support of ClearCase is the only data we know of that is relevant to the merging of parallel versions.

While there is no direct data about the number of components on average involved in the evolution of Windows NT, there is data provided about the specific case of fixing faults [4, page 319]: each fault repair usually required changing 3 to 5 files.

In both cases, the data as published is anecdotal in nature. An empirical study on parallel changes is necessary if we are to evaluate the scalability of a certain solution, to weigh the tradeoffs in costs, and to understand why and in what situation one approach may be better than another.

3 STUDY CONTEXT

This study is one of several strands of research being done in the context of the Code Decay Project [5], a multi-disciplinary and multi-institution project supported in part by NSF. It was conducted in part to understand how software systems evolved over time and how parallel changes played a role in that evolutionary process, to the extent that can be deduced from the change management data left behind over several decades.

We describe first the characteristics of the subsystem under study, then the change and configuration management data available to the Code Decay Project, and finally the change and configuration management processes.

3.1 The Subsystem Under Study

The data for this study comes from the complete change and quality history of a subsystem of the Lucent Technologies' 5ESS. This data consists of the change and configuration management history representing a period of 12 years from April 1984 to April 1996. This subsystem is one of 50 subsystems in 5ESS. It was built at a single development site. The development organization has undergone several changes in structure over the years and its size has varied accordingly, reaching a peak of 200 developers and eventually decreasing to the current 50 developers. There are two main product offerings, one for US customers and another for international customers. Historically, the two products have separate development threads although they do share some common files.

3.2 The 5ESS Change Management Process

Lucent Technologies uses a two-layered system for managing the evolution of 5ESS: a change management layer, ECMS [23], to initiate and track changes to the product, and a configuration management layer, SCCS [20], to manage the versions of files needed to construct the appropriate configurations of the product.

All changes are handled by ECMS and are initiated using an *Initial Modification Request* (IMR) whether the change is for fixing a fault, perfecting or improving some aspect of the system, or adding new features to the system. Thus an IMR represents a problem to be solved and may encompass the implementation of all or part of a feature. Features are the fundamental unit of extension to the system and each feature has at least one IMR associated with it as its problem statement.

Each functionally distinct set of changes to the code made by a developer is recorded as a *Modification Request* (MR) by the ECMS. An MR represents all or part of a developer's contribution to the solution of an IMR. (A developer may split his solution into multiple MRs if it appears to encompass multiple logical changes.) Multiple MRs may be needed to solve an IMR, especially if multiple developers are involved. A variety of information is associated with each IMR and MR. For example,

for each MR, ECMS includes such data as the date it was opened, its status, the developer who opened it, a short text abstract of the work to be done, and the date it was closed.

When a change is made to a file in the context of an MR, SCCS keeps track of the actual lines added, edited, 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 where it belongs.

The process of implementing an MR usually goes as follows:

1. Make a private copy of necessary files,
2. Try out the changes within the private copy,
3. When satisfied, retrieve the files from SCCS, locking them for editing,
4. Commit the changes as deltas in the SCCS, releasing the locks,
5. Retrieve the files again from the SCCS for reading,
6. Put the files through code inspection and unit testing,
7. Submit the MR for load integration and feature and regression test

There are several observations. In step 3, the developer has to make sure that his changes do not conflict with other recent changes put into the code. In step 6, the code that is inspected contains only the officially approved base code plus changes from the developer's MR. It does not include unapproved changes made by other developers. Hence the inspection and testing may not catch any conflicts when all these different MRs are combined. It is hoped that any conflicts are caught during load integration and feature and regression testing.

When all the changes required by an MR have been made, the MR is *closed* after all approval has been obtained for all the dependent units. Similarly, when all the MRs for an IMR have been closed, the IMR itself is closed, and when all IMRs implementing a feature have been closed the feature is completed.

4 DATA AND ANALYSIS

The change management data provides various different viewpoints from which to delineate the boundaries of, and to understand the nature of, the phenomena of parallel changes. We first discuss the different levels at which parallel development takes place, and then explore the effects of parallel changes at the file level and discuss the basic problem of change interference. We conclude this section by analyzing and summarizing the data about parallelism at the levels of features, IMRs, MRs and files.

In this section we make liberal use of histograms to provide a clear picture of the data that would not be evident if we were to report merely the minimum, mean, and maximum of each distribution. It is important to notice that the tails of several distributions are long and fall off more slowly than the Poisson or binomial distributions (classical engineering distributions). This is extremely important to consider in designing tools: if a tool is designed around the mean value, it will not be particularly useful for the critical cases that need the support the most, namely, those cases represented by the tail of the distribution.

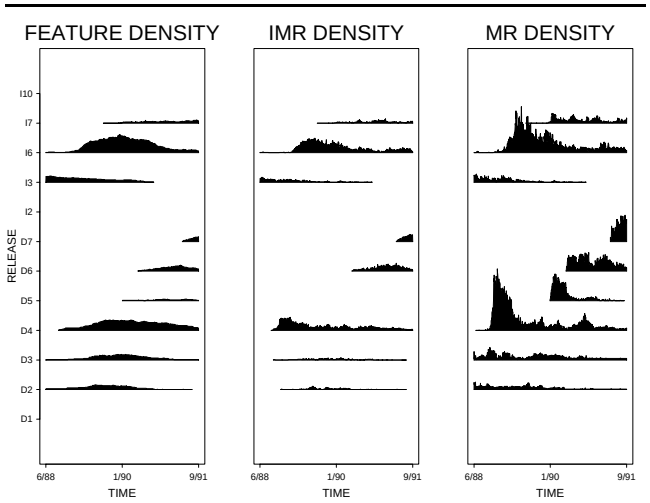

Figure 2: **Concurrent development activities in the development interval of Release I6.** These panels show the activities being conducted in parallel at the feature, IMR, and MR levels during the development interval for Release I6. Release I6 has peaks of approximately 60 open features, 50 open IMRs and 100 open MRs. The panels also show activities for other releases during the same time period.

4.1 Levels of Parallel Development

The 5ESS system is maintained as a series of releases, with each release offering new features on top of the existing features in previous releases. The timeline on Figure 1 shows the number of deltas applied every month to each release of the 5ESS subsystem under study. The top half shows the international releases (labeled I1–I15) and the bottom shows the domestic ones (labeled D1–D12). It shows that for each product line, there may be 3–4 releases undergoing development and maintenance at any given time.

Within each release shown in Figure 1, multiple features are under development. The overlapping time schedule of successive releases suggest that features for different releases are being developed almost concurrently. Figure 2 is a timeline showing the density of new feature development during the development interval of Release

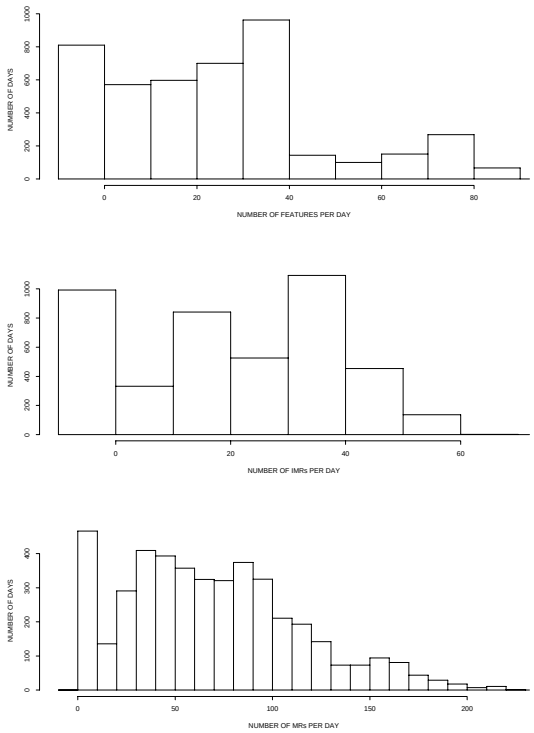

Figure 3: **Feature, IMR, MR distribution per day.** These histograms show the distribution of open features, IMRs, and MRs per day over the 12-year period under study.

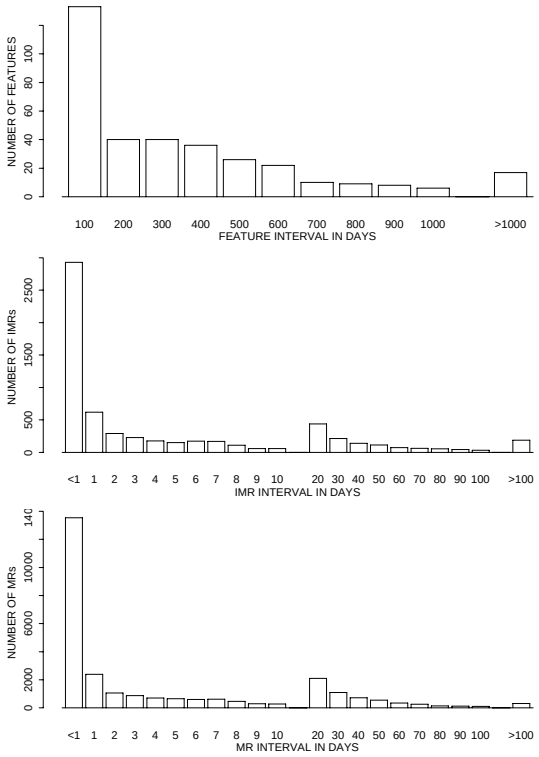

Figure 4: **Interval distributions.** These histograms show the development interval distributions for features, IMRs and MRs in number of days.

I6.[3] At its peak, there was work on about 60 features. It not only shows that multiple features are being developed concurrently for Release I6, but also shows that 8 other releases are doing new feature development.

Figure 2 also shows the density of IMRs and MRs developed for Release I6 as well as other releases in the same interval. At its peak, there were approximately 50 open IMRs and 100 open MRs.

Thus, we have parallel development going on at different levels in the development of this subsystem. Releases are being built in parallel with varying amounts of overlapping development. Features are being developed in parallel both within a single release and in the context of multiple releases. Typically multiple IMRs are being developed in parallel for each feature, and MRs are developed in parallel for each IMR. And, finally, files are changed in parallel within MRs, IMRs, features and releases.

[3]We picked Release I6 as an example here because it exhibited a high degree of parallel development at multiple levels and thus, was useful for our illustrations.

4.2 Multilevel Analysis of Parallel Development

To understand the amount of parallelism going on at the different levels, we examine the number of features, IMRs and MRs being developed per day. We then look at four measures associated with the amount of work within each feature, IMR and MR: their intervals, the number of files affected, the number of MRs involved, and the number of developers involved. Table 1 summarizes these data.

Figure 3 shows the frequency distributions of features, IMRs, and MRs being worked on per day. The feature and IMR distributions have means of 25 and 22, and maximum values of 86 and 62, respectively. On the other hand, there is a mean of 69 MRs open per day, and a maximum of more than 200. Note that in all cases the tail is very long with respect to the mean.

Figure 4 shows the frequency distributions of development intervals at the three levels. The intervals are measured by taking the dates of the first and last delta associated with that feature, IMR, or MR, and computing the difference. Thus the interval reflects the activity only with respect to coding.[4] One observation here is

[4]For instance, the feature interval measured excludes other

	Features				IMRs				MRs			
	Min	Median	Mean	Max	Min	Median	Mean	Max	Min	Median	Mean	Max
Active per day	0	23	25.3	86	0	21	21.8	62	1	65	69.3	223
Interval (days)	1	201	318.5	3344	< 1	1	14.6	2233	< 1	1	10.1	2191
#Files	1	8	31.0	906	1	1	4.3	388	1	1	1.1	15
#MRs	1	6	34.6	2188	1	1	2.6	86	n/a	n/a	n/a	n/a
#Developers	1	2	4.0	98	1	1	1.1	9	n/a	n/a	n/a	n/a

Table 1: **Data summary.** This table summarizes the data to be used in analyzing the degree of parallelism.

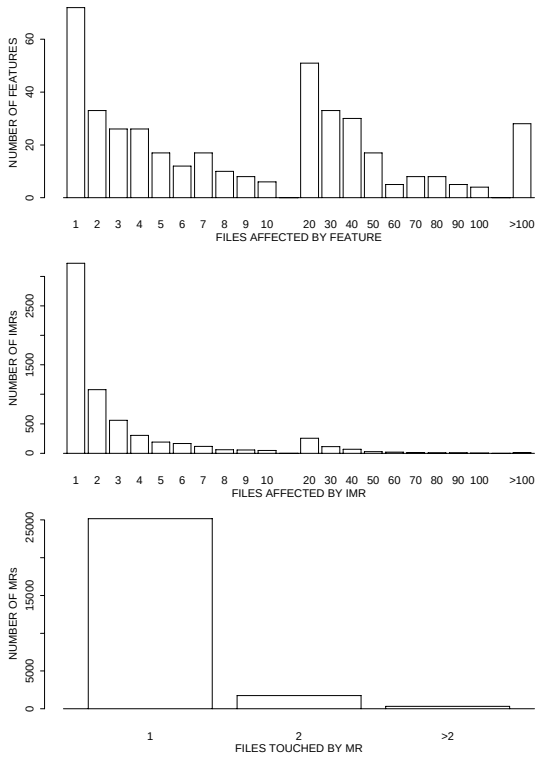

Figure 5: **Files touched.** These histograms show the distributions of number of files affected per feature, IMR and MR.

that the shapes of all three distributions appear to be similar, even though their scales are orders of magnitude apart. Also, 46% of the IMRs and 50% of the MRs are opened and solved on the same day. Nevertheless, the tails here are even longer with respect to the mean than in Figure 3.

Figure 5 shows the frequency distributions on the number of files affected in implementing each feature, IMR or MR. The number of files per feature exhibits a very large tail distribution, 33% of the features affected more than 20 files. On the other hand, 51% of the IMRs and more than 90% of the MRs affect only one file.

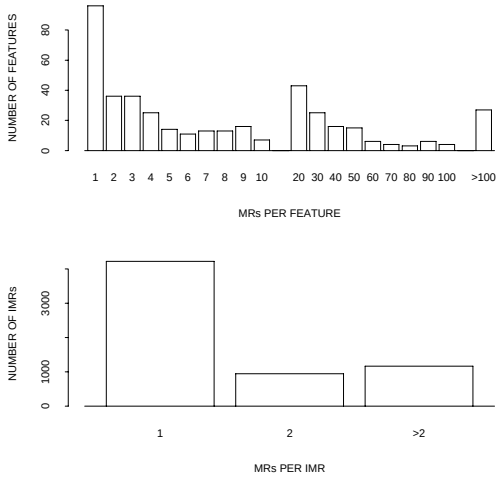

Figure 6: **Number of MRs used.** These histograms show the distributions of number of MRs used in implementing each feature and IMR.

Figure 6 shows the frequency distributions on the number of MRs it took to implement each feature and IMR. The number of MRs per feature again exhibits a large tail, 25% of the features needed more than 20 MRs. The tail for IMRs, while not as long as that for features, is still significant with a maximum of 86 needed for the largest IMR while the mean is less than 3.

Figure 7 shows the frequency distributions on the number of developers working on each feature and IMR.[5] The number of developers working on a feature does not have as large a tail as the number of MRs per feature, but there were still more than 20 features which involved more than 10 developers, with the largest feature involving 98 developers. Similarly, the mean is 1.1 developer per IMR, but the tail stretches out to a maximum of 9. Note however, the percentage of IMRs requiring more than one developer is only 10%.

feature activities like estimation, planning, requirements, design, and feature test.

[5]Because of the way MRs are defined, there can be only one developer per MR.

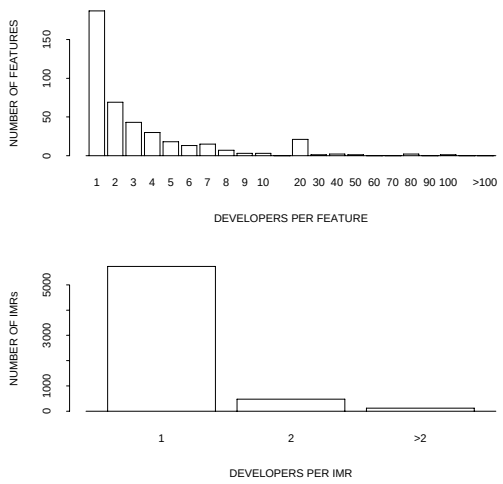

Figure 7: **Number of developers involved.** These histograms show the distributions of number of MRs used in implementing each feature and IMR.

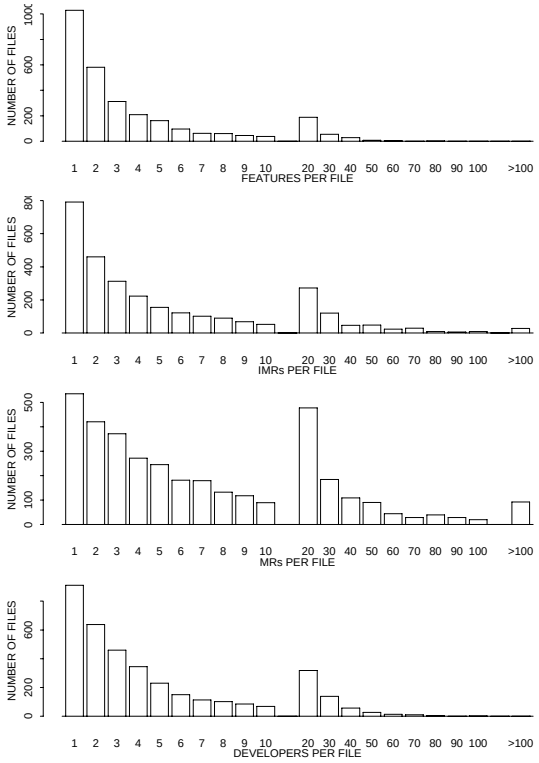

Figure 8: **Number of features, IMRs, MRs and developers per file.** These histograms show the distribution of the number of features, IMRs, MRs and developers affecting each file over the lifetime of the file.

4.3 Parallel Development Within a File

The preceding discussion does not show how these parallel activities interact with each other, particularly in the case when several of them make changes to the same file. Figure 8 shows the distribution of the number of features, IMRs, developers and MRs affecting each file over the lifetime of the file.

To illustrate further, Figure 9 shows the different levels of ongoing activity for a certain file. This clearly shows that parallel activities are going on at every level.

4.4 Parallel Versions

The set of changes belonging to a feature, IMR, MR and developer can be thought of as creating different versions of the code. Among these, MRs are the atomic component. Hence, in the subsequent discussion, we will use parallel MR activity as the basic unit of parallel development.

In Figure 10, we see that in the interval when Release I6 was being developed, about 60% of the files are touched by multiple MRs. Note also that the tail of the distribution is significant here — 17% of the files are touched by more than 10 MRs.

Figure 11 is a closeup of Figure 9. It magnifies the MR panel at one period with high activity. It shows that at one time, there were as many as 8 open MRs affecting this file, with 4 of them having deltas on the same day. We define PC_{max}, the maximum number of concurrently open MRs per day over the entire lifetime of the file, as our initial measure of the degree of parallel change. (For the file in Figure 9, $PC_{max} = 9$.) We computed PC_{max} for each file in the subsystem. Figure 12 shows the frequency distribution of PC_{max}. It shows that an average file may have up to 2 MRs per day, which translates to 2 active variants at a given time. It also shows that 55% the files never have more than one MR at a time, although about 25% of the files can have 2 MRs per day and 20% of the files can have 3 to 16 MRs per day.

5 EFFECTS OF PARALLEL CHANGES

In the preceding section we presented the phenomena of parallel changes in the context of a very large scale development. In this section, we look at the consequences of parallel development such as is found here. We first investigate the quality consequences of this parallelism and show how the higher the degree of parallelism, the higher the number of defects. We then look at one of the possible root causes of these quality problems, interfering changes, and discuss the most obvious cases of interference: changes on top of previous changes, and changes made within very close temporal proximity to each other.

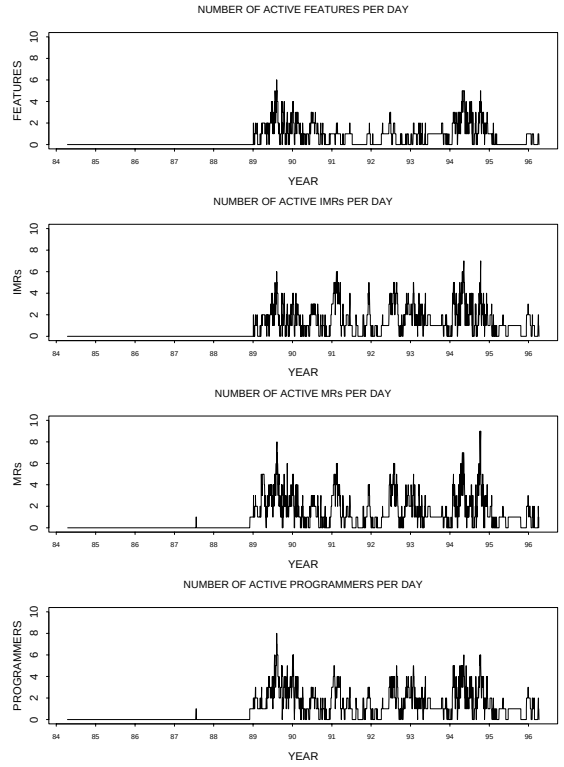

Figure 9: **Activity profile for one file.** The top two panels show the number of features and IMRs that affect this file over time. The third panel shows the number of open MRs modifying this file. The fourth panel shows the number of developers with open MRs modifying this file.

5.1 Implication on Quality

To examine the impact of parallel changes on software quality, we examined the defect distribution of the files for each value of PC_{max}. We counted as a defect every MR whose purpose is to correct a problem in the file. The MR classification was done automatically by analyzing the MR descriptions for known keywords. The paper by Mockus and Votta [17] describes the MR classification method in more detail.

In order to avoid double-counting the MRs, we recomputed the parallel development measure including only MRs opened up to 1994 and we plotted it against the number of defects discovered from 1994 to 1996. The results are shown in the boxplot in Figure 13.[6] The

[6] Boxplots are a compact way to represent data distributions. Each data set is represented by a box whose height spans the central 50% of the data. The upper and lower ends of the box marks the upper and lower quartiles. The data's median is denoted by a bold point within the box. The dashed vertical lines attached to the box indicate the tails of the distribution; they extend to the standard range of the data (1.5 times the inter-quartile range). The detached points are "outliers" lying beyond this range [3].

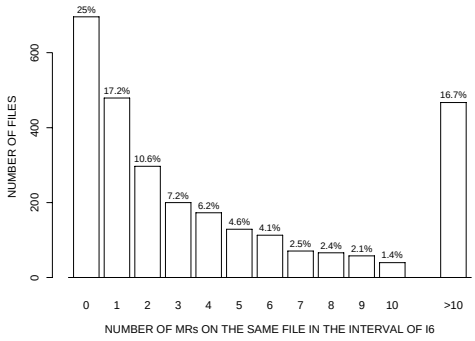

Figure 10: **Distribution of number of MRs touching each file in the development interval of Release I6.** Bar N shows the number of files which were touched by N MRs during the development interval of Release I6.

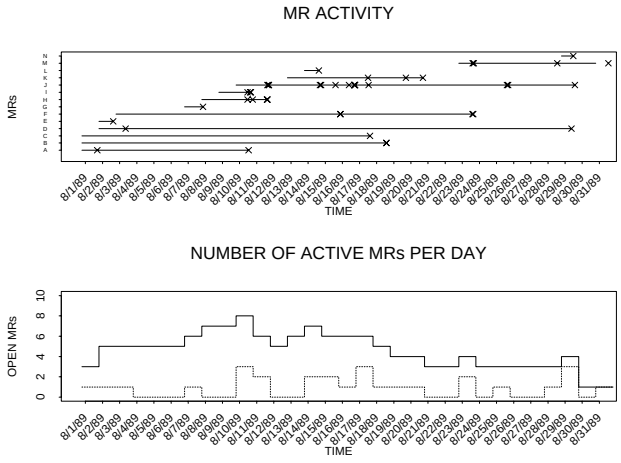

Figure 11: **A closer look at MR activity.** This is a closer look at the MR activity during one busy period (8/89) of the file in Figure 9. Each line in the top panel shows shows the lifespan of an MR being worked on during this period, from the date it was opened until the date it was closed. The X's indicate when deltas were made into the file. The solid line in the bottom panel shows the number of open MRs on each of those days. It is a magnification of the MR panel from Figure 9. The dotted line shows the number of deltas actually made on each day.

plot shows that files that have high degrees of parallel changes also tend to have more defects.

We then performed an analysis of variance (ANOVA) [1, Ch. 6] to account for the effects of other factors believed to contribute to the likelihood of increasing the

	Degrees of Freedom	Sum of Squares	Mean Squares	F Value	Significance
Number of deltas	1	14379	14379	3288	0.0000
File lifetime	1	4	4	1	0.3298
Creation date	1	548	548	125	0.0000
File size	1	164	164	38	0.0000
Past faults	1	241	241	57	0.0000
Parallel changes (PC_{max})	1	213	213	50	0.0000
Residuals	3266	13936	4		

Table 2: **Analysis of variance.** This table shows the contributions of various factors to the variance in number of defects. The column of major interest is the last one, which gives the significance of the contribution of each factor to the variance. As shown here, every factor is significant except lifetime.

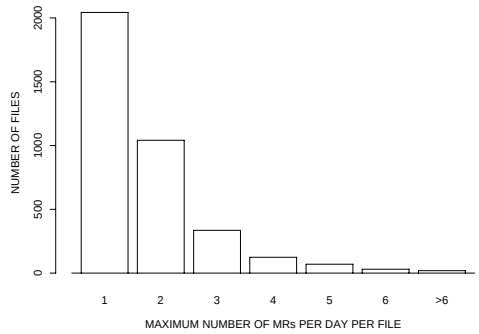

Figure 12: **Maximum number of parallel MRs per file.** This histogram shows the distribution of PC_{max}, the maximum number of parallel MRs that affected each file in a day.

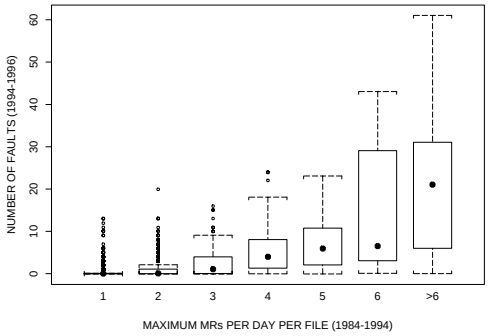

Figure 13: **Parallel development (PC_{max}) vs. number of defects.** This boxplot shows the number of defects for each file, grouped by degree of parallel changes.

number of defects [8]. We examined the following: file creation date (date the first delta was made), lifetime of file (from file creation date up to 1994 or the date of last delta, whichever came first), total number of deltas made between 1984-1994, size of file at 1994 and past faults found in the file between 1984 - 1994. The results are shown in Table 2. The sum of squares and corresponding significance probabilities were computed from the first factor to the last. The table shows that, even after accounting for all of these other factors, the degree of parallel changes PC_{max} makes a significant contribution to the variance of the defect distribution. (See Appendix A for more detailed explanation.)

We also ascertained that the results were not an artifact of the measure of degree of parallel changes that we defined. We had been using PC_{max}, the maximum number of MRs open in parallel, as our degree of parallel changes (e.g., for the file in Figure 9, $PC_{max} = 9$). Another measure of the degree of parallel changes is to count, for each file, the number of days in which more than one MR was open. We label this as PC_{days}. Figure 14 shows the distribution of PC_{days} against defect count (cor = 0.63). When we replaced PC_{max} with PC_{days} in the ANOVA model, the results remained significant.

Yet another measure of parallel changes is to take the number of days with more than one open MR and weigh each day by the number of open MRs. We label this as PC_{wdays}. Figure 15 shows the distribution of PC_{wdays} against defect count (cor = 0.62). When we used PC_{wdays} in the ANOVA model, the results again remained significant. Table 3 compares the three measures of the degree of parallel changes. Of these three, PC_{days} appears to be the best measure.

5.2 Interfering Changes

Thus far, we have examined the amount of parallel activities going on and how it might contribute to quality problems. We have not actually delved into the mechanisms by which parallel changes could cause defects.

	Degrees of Freedom	Sum of Squares	Mean Squares	F Value	Significance
Maximum parallel MRs (PC_{max})	1	213	213	50	0.0000
Number of days with parallel MRs (PC_{days})	1	705	705	171	0.0000
Weighted number of days (PC_{wdays})	1	551	551	132	0.0000

Table 3: **Three measures of degree of parallel changes.** This table compares the contribution of the three measures of the degree of parallel changes. The sum of squares, F values and significance values are obtained when each one replaces the parallel changes entry in the ANOVA table in Table 2.

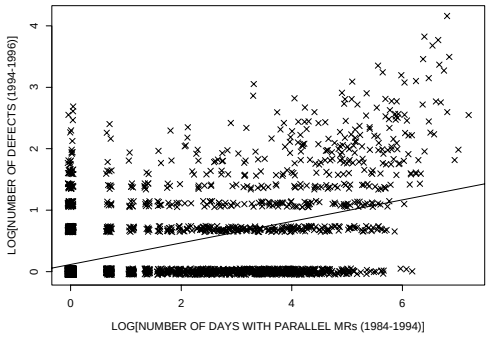

Figure 14: PC_{days} **(number of days with parallel MRs) vs. number of defects.** This scatterplot shows the number of defects for each file, plotted against PC_{days}, the number of days the file had parallel MRs. A log transformation was applied to both axes to spread the points. In addition, a small random offset was added to each point to expose overlapping points.

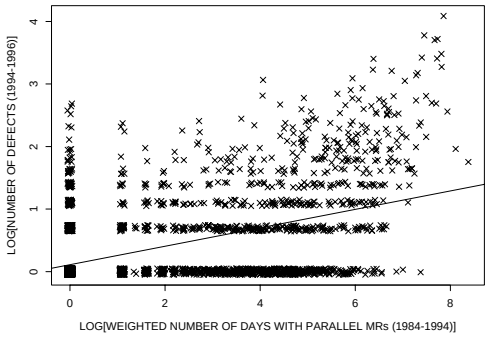

Figure 15: PC_{wdays} **(weighted number of days with parallel MRs) vs. number of defects.** This scatterplot shows the number of defects for each file, plotted against PC_{wdays}, the number of days the file had parallel MRs, weighted by the number of parallel MRs per day. Transformations were applied as in Figure 14.

In this section, we provide results of our initial investigation into parallel changes that interfere with each other.

Upon analyzing the delta data, we found that 12.5% of all deltas are made to the same file by different developers within 24 hours of each other. Given this high degree of parallel development, it seems likely that changes by one developer will interfere with changes made by another developer. For this study we have looked at the *prima facie* case where changes interfere by one change physically overlapping another. For example, Figure 16 traces several versions of the file examined in Figure 9 as 5 deltas were applied to it during a 24-hour period. Developer A made 3 deltas, the first two of which did not affect this fragment of code. Then developer B put in changes on top of A's changes. Finally some of B's changes were modified by developer C on the same day.

Across the subsystem, 3% of the deltas made within 24 hours by different developers physically overlap each others' changes. Note that physical overlap is just one way by which one developer's changes can interfere with others. We believe that many more 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.

6 VALIDITY

In any study, there are three aspects of validity that must be considered in establishing the credibility of that study: construct validity, internal validity, and external validity. We consider each of these in turn.

We have operationalized the definition of parallel changes in several ways. First, we looked at the level of parallel development with respect to different levels (release, feature, IMR, MR and file). Second, in deriving a summary measure of the degree of parallel development for use in the quality model, we looked at multiple measures and showed that they are consistent with each other. Thus we argue that we have the necessary construct validity.

As can be seen from the data as we have presented it,

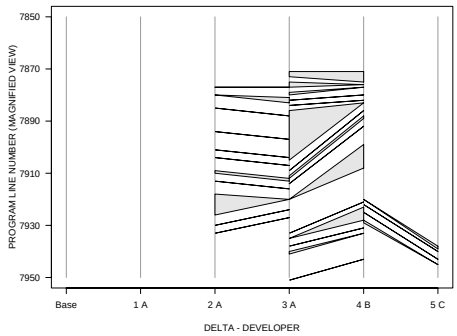

Figure 16: **Lines changed per delta.** Each vertical line represents a version of the file as it was changed by a delta (denoted 1–5). The x-axis also encodes the developer who made the delta (denoted A–C). The delta sequence is read from left to right. The lines connecting the vertical lines show where lines have been changed from one version to the next. Lines that diverge show where new code was inserted while lines that converge show where code was deleted. The trapezoids show where code was changed. Note that this figure only shows a fragment of each program version, approximately from lines 7850-7950.

we have done only the minimal amount of data manipulation and then only to put it into easily understood forms of summarization. Also, in the quality study, we have sought to account for other factors that may affect the number of defects in the software. Thus we argue that we have the necessary internal validity.

It is in the context of external validity that we must be satisfied with arguments weaker than we would like. We argue from extra data (namely, visualizations of the entire 5ESS system similar to Figure 1) that this subsystem is sufficiently representative of the other subsystems to act as their surrogate.

The primary problem then is the representativeness of 5ESS as an embedded real time and highly reliable system. In its favor are the facts that it is built using a common programming language (C) and development platform (UNIX). Also in its favor are the facts that it is an extremely large and complicated system development and that problems encountered here are at least as severe as those found in smaller and less complicated developments. Finally, it follows an ISO 9000 compliant process that has consistently received a CMM [18] rating of 2 and satisfies most of the process areas for levels 3 and 4, the same as, if not better than, a good number of software organizations in the industry. Thus we argue that our data has a good level of external validity and is generalizable to other developments of similar domains.

7 SUMMARY AND EVALUATION

7.1 Study Summary

This work represents initial empirical investigations to understand the nature of large scale parallel development. The data shows that in this subsystem:

- 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 may interfere with each other.
- Over the interval of a particular release (I6), the number of files changed by multiple MRs is 60% which, while not directly concurrent, is concurrent with respect to the release. These may also have 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.
- There is a significant correlation between files with a high degree of parallel development and the number of defects. Moreover, even accounting for lifetime, size and number of deltas, the degree of parallel changes makes a significant contribution to the variance of the defect distribution.

The data presented illustrates the problems of evolution, scale, and multiple dimensions of organization described in Section 1: of evolution because of the increasing number of releases that needed to be maintained; of scale because of the sheer number of parallel activities going on within a release interval; and of organization because the parallel activities are not independent, but that at some point they need to coordinate, especially if they are modifying common files.

7.2 Evaluation of Current Support

As we mentioned in subsection 4.2, the histograms provide a critical picture of the problems that need to be solved. In particular, the tails of the distributions are the significant factors to consider in technical support, not the mean values. In both the cases of workspaces and merging, we claim that those critical factors have not been understood or appreciated.

The data in subsection 4.2 suggests that, if each MR had its own workspace, we would need on the order of 70 to 200 workspaces per day for this particular subsystem. (And this is just one of 50 5ESS subsystems!) Moreover, since 50% of MRs are solved in less than a day,

the cost and complexity of constructing and destroying workspaces becomes very important. One might reduce the number of workspace per day by assuming one workspace per IMR or per feature. Doing so introduces further coordination problems since there may be more than one developer working on the IMR or feature.

Given the multi-level nature of feature development, one might imagine the need for a hierarchical set of workspaces[12] such that there is a workspace for each feature, a subset of workspaces for each IMR for that feature and then individual workspaces for each MR. In either case, further studies are needed to determine the costs and utility of workspaces in supporting the phenomena we have found in this study.

The utility of the current state of merge support depends on the level of interference versus non-interference. The data in subsection 4.4 indicates that about 45% of the files can have 2 to 16 parallel versions with potentially interfering changes. It is not clear how well current merge technologies will be able to support this degree of parallel versions — how do you merge 16 parallel versions? The data we have uncovered certainly leads us to be sympathetic with Adele's claim that frequent updates are necessary for coordinated changes and that waiting until commit time will lead to parallel versions that cannot be merged without some very costly overhead and coordinated effort. In fact, their supported strategy is what is left unsupported in these developments reported here.

Further studies are needed to assess the validity and utility of merge technologies. We note in the next section one such study that will help to assess this area.

The *synchronize and build* strategy poses a problem in this context where features are the primary unit of work. Features represent a set of logically coherent changes to the system. As noted in Figure 5, features frequently involve a large number of files, with 33% of all features affecting more than 20 files and a maximum of 906 files as shown in Table 1. The current build process synchronizes at the MR level. However, each MR represents only a partial solution to a problem and failure to include all the dependent MRs has been a common cause of build problems. The same problem would also arise at the IMR level because IMRs sometimes depend on MRs belonging to other IMRs. Further studies are needed to understand the optimal build strategy.

7.3 Process and Project Management

Because of the direct correlation between the degree of parallelism and an increased number of defects, process and project management need to take a careful look at how to support the development process at this particular point. A study of the software development organization which maintains this subsystem yielded two results relevant to parallel development, 1) a focus toward development interval reduction by gradually shifting from a code ownership model – in which a developer was designated to be the "owner" of one or more modules of code – to a feature ownership model – in which a developer was authorized to make all the changes necessary to implement a given feature or fix, and 2) a trend toward features that cut across an increasingly larger number of modules [21]. While the orientation towards feature development has useful properties for evolving and marketing the product, the resulting parallel development by multiple developers compared to that with file ownership poses significant problems that need to be carefully managed.

As it is very likely that the changes done in parallel conflict with each other, it is very important that the developers making the concurrent changes understand what each other is doing and how their changes interact with each other. This is the area where tool support is needed. Where these interdependencies cannot be managed automatically, they must be managed manually.

Much of the current coordination is done informally between developers where they know there are conflicts. The conceptual distance between the changes exacerbates the problem and increases the need for explicit coordination — that is, developers working on the same IMRs are likely to understand how the changes fit together much better than those working on different features in different releases.

7.4 Contributions

In our observational case study, we have established that

- parallel development is a significant factor in large–scale software development;
- current tool, process and project management support for this level of parallelism is inadequate; and
- there is a significant correlation between the degree of parallelism and the number of defects.

In addition, we have provided a novel form of visualization for the differences within a sequence of versions of a file, showing where code has been inserted, deleted and replaced (see figure 16).

7.5 Future Directions

We have looked at only the *prima facie* conflicts, namely, those where there are changes on changes or changes within a day of each other. A more interesting class of conflicts are those which we might term *semantic conflicts*. These cases arise where changes are made to the same slices of the program and hence may interfere with each other logically rather than syntactically.

This phenomena requires us to look very closely at the files themselves rather than just the change management data. Our plans for this analysis include combining the use of dataflow and slicing analysis techniques to determine when semantic interference occurs.

Given the appropriate analysis techniques, we will then look at a subset of the files to determine the degree of interference associated with various degrees of parallelism and to establish the correlation with the existing defect data.

ACKNOWLEDGEMENTS

The Code Decay Project [5] is a multi-disciplinary and multi-institution project for which a common infrastructure has been created in support of multiple strands of software engineering research. We thank several members of the project, Audris Mockus and Todd Graves, who extracted and prepared the change data into a form we could use. Todd Graves was supported by NSF Grant SBR-9529926. We also thank Dave Atkins who helped us to better understand the current 5ESS software development process.

REFERENCES

[1] George E. Box, William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters*. John Wiley and Sons, Inc., 1978.

[2] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[3] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tuckey. *Graphical Methods For Data Analysis*. Chapman & Hall, 1983.

[4] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free Press, 1995.

[5] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. on Software Engineering*, 2000. To appear.

[6] Jacky Estublier and Rubby Casallas. The Adele configuration manager. In Walter F. Tichy, editor, *Configuration Management. Trends in Software*. John Wiley & Sons, 1994.

[7] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

[8] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering*, 2000. To appear.

[9] Rebecca E. Grinter. Doing software development: Occasions for automation and formalisation. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, Lancaster, U.K., September 1997.

[10] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. on Software Engineering and Methodology*, 11(3):345–387, July 1989.

[11] Charles M. Judd, Eliot R. Smith, and Louise H. Kidder. *Research Methods in Social Relations*. Harcourt Brace Jovanovich College Publishers, 1991.

[12] Gail E. Kaiser and Dewayne E. Perry. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Proceedings of the 1987 International Conference on Software Maintenance*, pages 108–114, Austin, Texas, September 1987.

[13] David B. Leblang. Personal communication.

[14] David B. Leblang. The CM challenge: Configuration management that works. In Walter F. Tichy, editor, *Configuration Management. Trends in Software*. John Wiley & Sons, 1994.

[15] Alex Mahler. Variants: Keeping things together and telling them apart. In Walter F. Tichy, editor, *Configuration Management. Trends in Software*. John Wiley & Sons, 1994.

[16] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) Switching System. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July–August 1985.

[17] Audris Mockus and Lawrence Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 2000 International Conference on Software Maintenance*, San Jose, CA, 2000. To appear.

[18] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model for software (version 1.1). Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, Feb. 1993.

[19] Dewayne E. Perry. System compositions and shared dependencies. In *6th Workshop on Software Configuration Management*, Berlin, Germany, March 1996.

[20] Marc J. Rochkind. The Source Code Control System. *IEEE Trans. on Software Engineering*, SE-1(4):364–370, December 1975.

[21] Nancy Staudenmayer, Todd Graves, J. Steve Marron, Audris Mockus, Dewayne Perry, Harvey Siy, and Lawrence Votta. Adapting to a new environment: How a legacy software organization copes with volatility and change. In *5th International Product Development Management Conference*, Como, Italy, May 1998.

[22] Walter Tichy. Design, implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, pages 58–67, Tokyo, Japan, September 1982.

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

[24] Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, 2nd edition, 1994.

A REVIEW OF ANOVA

This appendix gives some additional explanations of the ANOVA table in Table 2.

The first column gives the source of variation or the factors being considered.

The second column is the degrees of freedom, which is always 1 for numeric variables. For the residuals, it is the difference between the number of points in the analysis and the number of degrees of freedom used up by the factors being considered.

The third column (sum of squares) is obtained as follows:

1. for the first factor (e.g. number of deltas), fit a linear regression line between that factor and the number of faults, then sum up the squares of the differences between each of the files' number of faults and the corresponding value fitted from the regression;
2. for the second factor, fit a linear regression line between that factor and the residuals from the first regression and repeat the sum of squares calculation;
3. for the third factor, fit a linear regression line between that factor and the residuals from the second regression and repeat the sum of quares calculation;
4. and so on.

The fourth column is the mean square, which is just the sum of squares divided by the degrees of freedom.

The fifth column is the F statistic, which is the ratio of the mean square of each factor divided by the mean square of the residuals, and can be thought of as a measure of how real the contribution of each factor is relative to chance (the larger the number, the higher the likelihood of a real effect).

The last column is the significance of the F statistic (a value less than 0.05 is usually considered significant).

Note that because the factors are not all independent of each other and because of the way the sum of squares are computed, the significance values are sensitive to the ordering of the factors. In this case, we purposely put parallel changes at the end to see if it would still be significant after all the other factors have been considered. As it turned out, the contribution of parallel changes was significant regardless of its position in the ordering.

Introductory explanations of ANOVA can be found in most statistics textbooks. In addition, there are several online references:

1. David Stockburger. ANOVA: Why multiple comparisons using t-tests is not the analysis of choice. http://www.psychstat.smsu.edu/introbook/sbk27.htm,
2. David Lane. Partitioning the sums of squares. http://www.ruf.rice.edu/~lane/hyperstat/B83612.html.