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

Dewayne E. Perry
Software Production Research Department
Bell Laboratories
Murray Hill, MH 07974 USA
+1 908 582 2529
dep@research.bell-labs.com

Harvey P. Siy, Lawrence G. Votta
Software Production Research Department
Bell Laboratories
Naperville, IL 60566 USA
+1 630 224 6830, +1 630 713 4612
{hpsiy,votta}@research.bell-labs.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; and 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.

KEYWORDS
Change management, parallel/concurrent changes, configuration management, parallel versions, merging interfering and non-interfering versions

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 [1] 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] [14] 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 organizational structures, by project management, by process, and by technology? We are particularly interested in the problems of technology support.

Before we can adequately answer these questions we need to understand the depth and breadth of the problem. To explore the dimensions of this phenomena, we take a look at the history of a subsystem of Lucent Technologies' 5ESS® telephone switch [13] to understand the various aspects of parallel development in the context of a large software development organization and project.

* Research supported in part by NSF Grant SBR-9529926 to the National Institute of Statistical Sciences.

[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.

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 [18]. In this way, the method is similar to the ones used in Astronomy and the social sciences [8]. Finally, a theory is built using these observations (e.g., with grounded theory [5]) 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 two 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. We have several hypotheses for this subproblem. 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 likelyhood. 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 and discuss the construct, internal and external validity of our study. Finally, we summarize our findings, evaluate the various means of technological support in the light of our results, and suggest areas for further research and development.

2 RELATED WORK

In terms of technical support for parallel changes, there are two different strands of research that are relevant: configuration management and program analysis research.

2.1 Configuration Management

Classic configuration management systems in widespread use today, SCCS [15] and RCS [16], embody the traditional library metaphor where source files are checked out for editing and then checked back in[6]. They induce a sequential model of software development. The locking for an *edget* operation guarantees that only one user can change a particular file at a time and blocks other developers from making changes until an *edput* 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.

One of the standard features of even the classic configuration management systems that enables developers to create parallel versions is the branching mechanism. The problem, however, is not in creating parallel versions, but in figuring out how to merge them back into a single version. Mahler [12] makes a distinction between temporary and permanent variants. Permanent variants are "branches in the product development path that have their own life cycle". Temporary variants on the other hand are meant to be merged eventually and only need to exist for the time needed until merging.

PureAtria's ClearCase® [11] provides support for automatic merging of up to 32 versions. This support consists of automatically figuring out the best sequence for merging the changes.

> An *automatic merge* facility can help with the mechanics of merging source code changes. A merge too that knows the *common ancestor* of the versions being merged can generally merge with little or no human interaction. Experience with DSEE and with ClearCase has shown that over 90% of changed files can be merged without asking any questions. The merge tool asks the user to resolve a conflict in the other cases. In about 1% of the the merge tool inappropriately makes an automatic de-

> cision, but nearly all of those cases are easily detected because they result in compiler syntax errors. [11]

This data came from an in-house merge of the Windows port of ClearCase with their UNIX version [10]. The merge involved several thousand files resulting from nine to twelve months of diverging development effort by about 10 people.

The Adele Configuration Manager [4] incorporates the notion of *workspaces* into configuration management to provide support for change management. Within a workspace a lock can be set on a file which causes the transparent creation of a copy (referred to as *dynamic versioning*). Releasing the lock causes the merging of the dynamic copies. Coordination control is provided amongst the workspaces (WSs) because

> ... object merging is not a perfect mechanism. Inconsistencies may arise from an object merger; the probability of problematic mergers rapidly increases with the number of changes performed in both copies. Were mergers to be performed only at transaction commit, most of them would not be successfully performed. Frequent mergers, at some well defined points, are needed to maintain two cooperating WSs in synch. [4]

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.

2.2 Program Analysis

The other strand of research is that of Horwitz, Prins and Reps' [7] work on integrating noninterfering versions. 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.

2.3 Synchronize and Build

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 [14]. Cusumano and Selby [3] noted this problem in the course of applying Microsoft's *synch and build* strategy to Windows NT. Their solution to coordinating changes [3, 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.

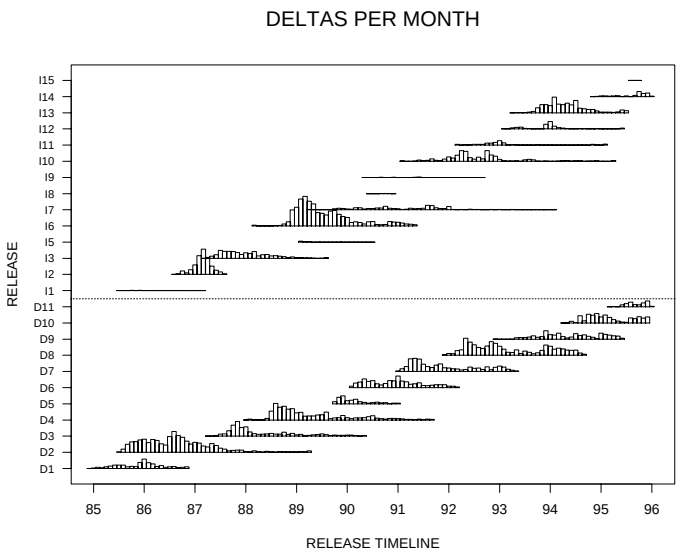

Figure 1: **Timeline of parallel releases.** Each histogram represents work being done for one release of the software. The top and bottom halves show releases for the international and domestic products, respectively.

This problem of coordinating changes is certainly an important one in the context of large scale systems build out of separately evolved components and of multiple dimensions of organization such as we have the system under study.

2.4 Empirical Evaluation

In neither Adele nor Reps, Prins and Horwitz is any data offered in support of their approaches. In the Adele case, we believe the motivation to have come from usage experience of the sequentialization of development. In the case of Horwitz, Prins and Reps, we believe the motivation to be that of advancing basic science by the investigation of an interesting but difficult problem.

The data offered in support of ClearCase is the only data we know of that is relevant to the merging of parallel versions and that data, as published, must be considered anecdotal.

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 bugs [3, page 319]: each bug fix required changes on average to 3 to 5 files.

3 STUDY CONTEXT

This study is one of several strands of research being done in the context of the Code Decay Project [2], a multi-disciplinary and multi-institution project sup-

ported by NSF.

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 restructuring 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 and Configuration Management Process

Lucent Technologies uses a two-layered system for managing the evolution of 5ESS: a change management layer, ECMS [17], to initiate and track changes to the product, and a configuration management layer, SCCS [15], 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 solve 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 change requests is recorded as a *Modification Request* (MR) by the ECMS. An MR represents all or part of a solution to a problem. 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, 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.

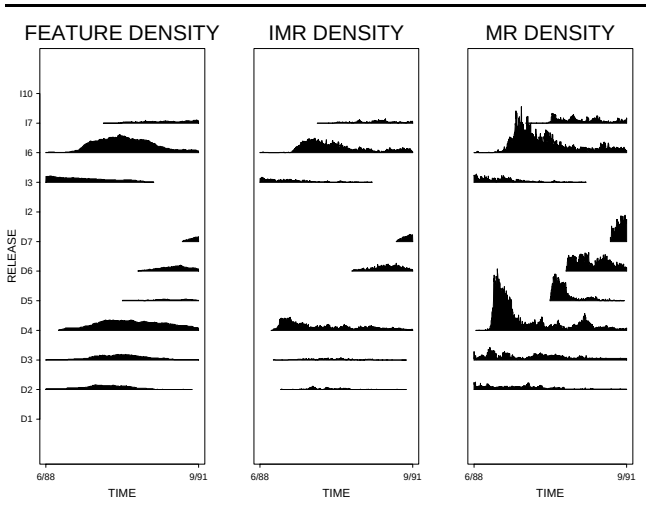

Figure 2: **Concurrent development activities in the development interval of release I6.** These panels show the activities being conducted in parallel at the feature, IMR, and MR levels during the development interval for release I6. It also shows activities for other releases during the same time period.

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. Commit the changes as deltas in the SCCS,
4. Put the private copy through code inspection and unit testing,
5. Submit the MR for load integration and feature and regression test

There are several observations. At any given time, there may be multiple private copies of a file being edited by different developers. Unless the developers are aware of each others' work, the changes being made by other developers are not visible until these developers submit their MRs for integration. 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.

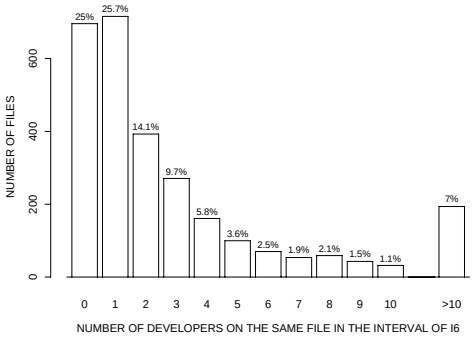

Figure 3: **Distribution of number of developers modifying each file in the development interval of release I6.** Bar N shows the percentage of files which were worked on by N developers during the development interval of release I6.

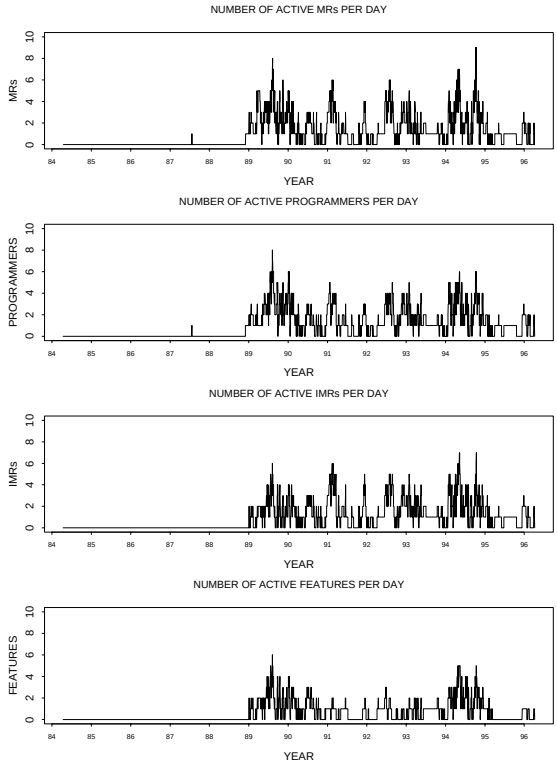

Figure 4: **Activity profile for one file.** The top panel shows the number of open MRs modifying this file over time. The second panel show the number of developers with open MRs modifying this file. The third and fourth panels show the number of IMRs and features, respectively, that are involved.

4 DATA AND ANALYSIS

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 I6. 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.

4.2 Effects of Parallel Development on a File

Figure 2 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. In Figure 3, we see that in the interval when release I6 was being developed, about 50% of the files are modified by more than one developer. Note also that the tail of the distribution is significant here — 25% of the files are modified by four or more developers.

To illustrate further, Figure 4 shows the various levels of activity going on for a certain file. This clearly shows that several developers may be working on the same file at the same time. Figure 5 is a closeup of the period with the highest activity. It shows that at one time, as many as 8 developers have open MRs affecting this file and as many as 4 modified the file on the same day.

4.3 Interfering Changes

Upon analyzing the available 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, sometimes changes by one developer may interfere with changes made by another developer by physically overlapping them. For example, Figure 6 traces several versions of the file examined in Figure 4 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

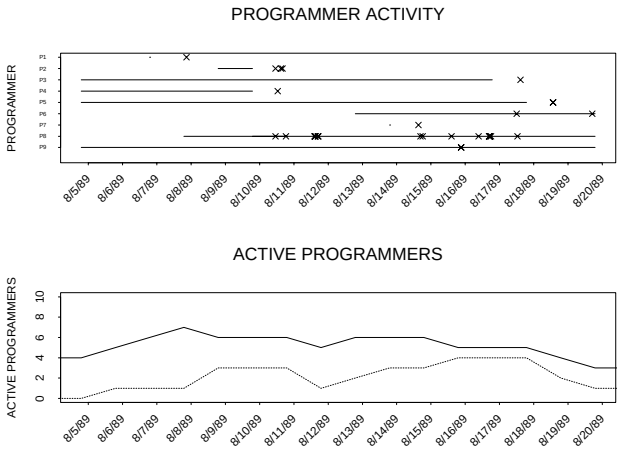

Figure 5: **A closer look at developer activity.** This is a closer look at the developer activity during the busiest period (8/89). Each line in the top panel shows MRs being worked on by 9 developers during this period. The X's indicate when they made deltas into the file. The solid line in the bottom panel shows the number of developers who have open MRs on each of those days. It is a magnification of the developer panel from the previous figure. The dashed line shows the number of developers who actually made deltas on each day.

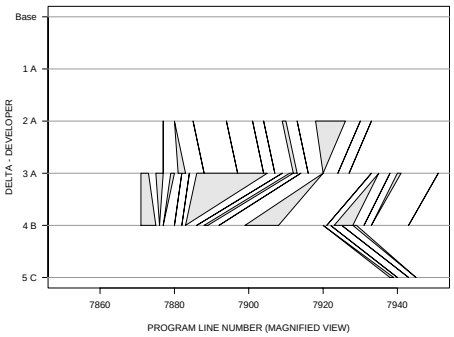

Figure 6: **Lines changed per delta.** Each horizontal line represents a version of the file as it was changed by a delta (denoted 1–5). The y-axis also encodes the developer who made the delta (denoted A–C). The delta sequence is read from top to bottom. The lines connecting the horizontal lines show where lines have been changed from one version to the next. An upright triangle shows where new code was inserted while an inverted triangle shows where code was deleted. The trapezoids show modifications of blocks of code. Note that this figure only shows a fragment of each program version, approximately from lines 7850-7950.

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.

4.4 Multilevel Analysis of Parallel Development

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.

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 7 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 70 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 8 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.[3] One observation here is that the shapes of all three distributions appear to be similar, even though their scales are orders of magnitude apart. Note also that 46% of the IMRs and 50% of the MRs are opened and solved on the same day. More importantly, the tails here are even longer with respect

[3]For example, the feature interval measured excludes other feature activities like estimation, planning, requirements, design, and feature test.

	Features			IMRs			MRs		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
Being worked on per day	0	25.3	86	0	21.8	62	1	69.3	223
Interval (days)	1	318.5	3344	< 1	14.6	2233	< 1	10.1	2191
Files affected	1	31.0	906	1	4.3	388	1	1.1	15
MR count	1	34.6	2188	1	2.6	86	n/a	n/a	n/a
Developer count	1	4.0	98	1	1.1	9	n/a	n/a	n/a

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

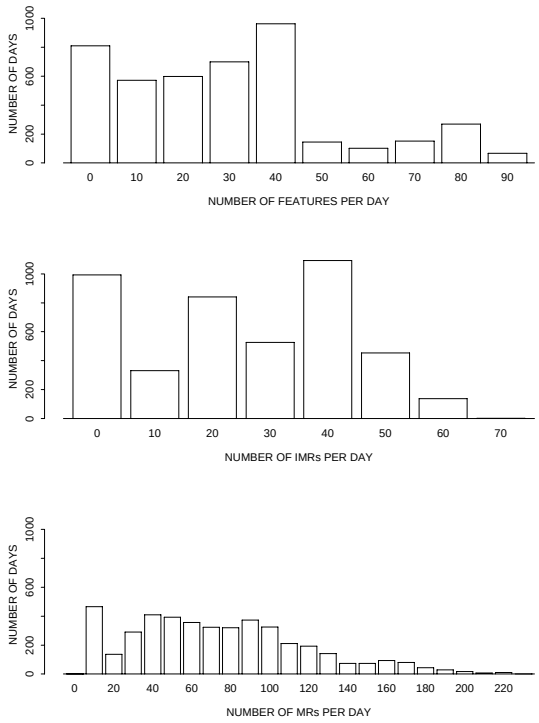

Figure 7: **Feature, IMR, MR distribution per day.** These histograms show the distribution of the number of features, IMRs, and MRs being worked on per day.

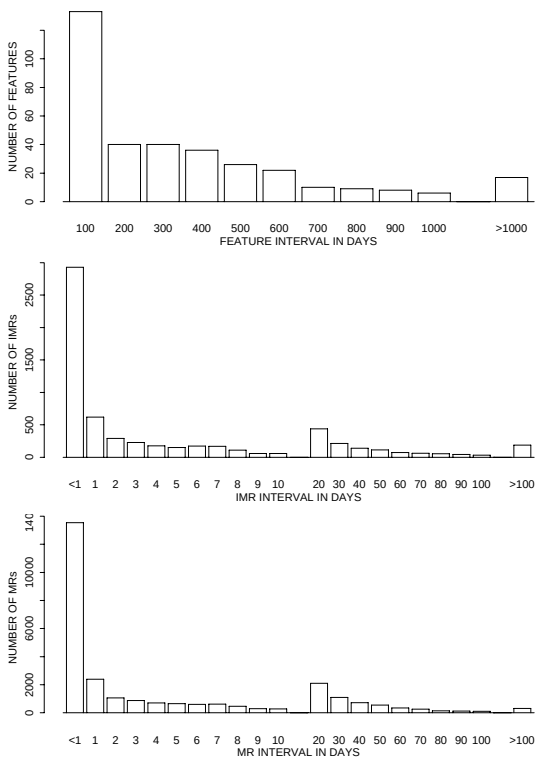

Figure 8: **Interval distributions.** These histograms show the development interval distributions for features, IMRs and MRs in number of days.

to the mean than in the previous figure.

Figure 9 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, with the maximum at 900. On the other hand, 51% of the IMRs and more than 90% of the MRs affect only one file.

Figure 10 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, with the maximum needing 2,000 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 11 shows the frequency distributions on the number of developers working on each feature and IMR.[4] The number of developers working on a feature does not have as large a tail as the number of MRs, but there were still some 20 features which involved 11 to 20 developers and the largest feature had 98 developers. Similarly, the mean is one developer per IMR, but the tail stretches out to a maximum of 9. Note however, the percentage of IMRs requiring more than one developer

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

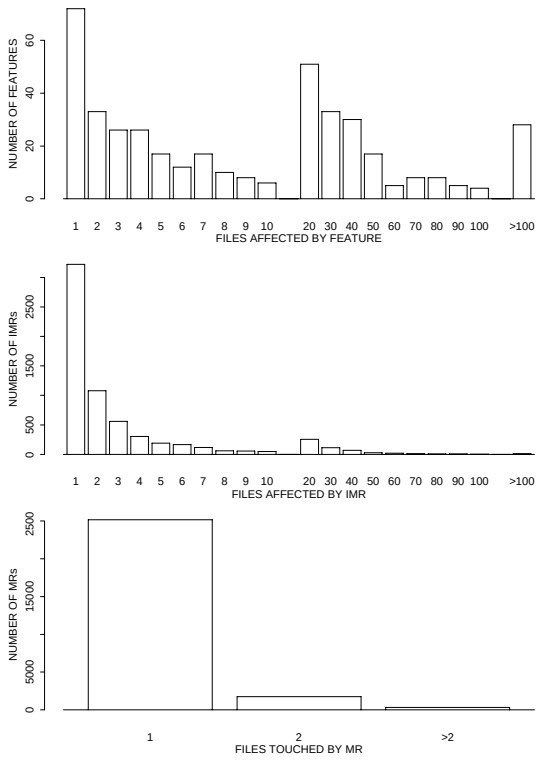

Figure 9: **Files touched.** These histograms show the distributions of number of files affected per feature, IMR and MR.

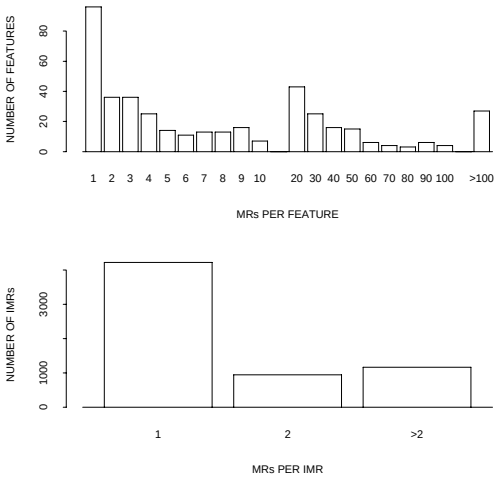

Figure 10: **Number of MRs used.** These histograms show the distributions of number of MRs used in implementing each feature and IMR.

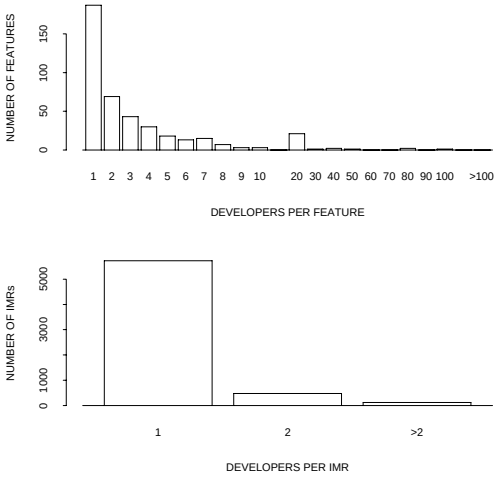

Figure 11: **Number of developers involved.** These histograms show the distributions of number of MRs used in implementing each feature and IMR.

is only 10%.

4.5 Parallel Versions

Figure 12 shows the distribution of the number of MRs affecting each file over the lifetime of the file. This is an unreasonably high upper bound for the number of versions for a file. A tighter upper bound would be to get the maximum of the number of active MRs per file (e.g. 8 for the file in Figure 4). Figure 13 shows the distribution of this quantity over all files. It shows that an average file may have up to 1.7 MRs per day, which translates to 1.7 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 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.

In trying to understand the phenomena of parallel changes it has been necessary to understand it at the various levels at which it occurs: the product level, the individual release level, the problem (IMR) level and the solution (MR) level. The measures that we have taken at these levels are precisely those which provide us with the critical information about parallelism of development. Thus we argue that we have the necessary construct validity.

As can be seen from the data as we have presented it, we have done only the minimal amount of data manipulation and that to put it into easily understood forms

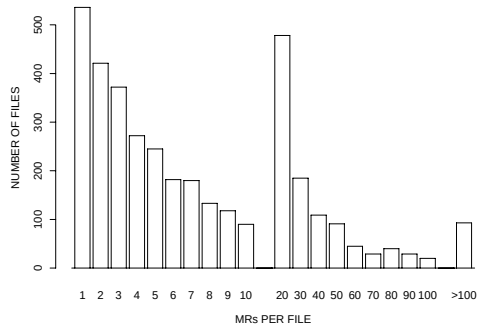

Figure 12: **Number of MRs per file.** This histogram shows the distribution of the number of MRs affecting each file over the lifetime of the file.

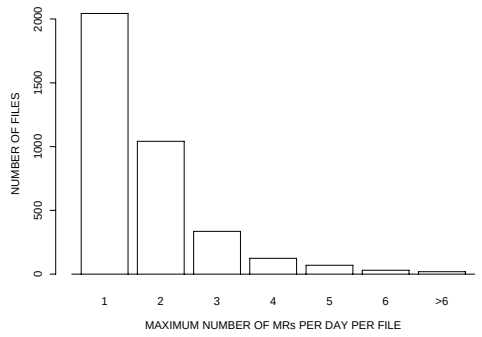

Figure 13: **Maximum number of MRs per file per day.** This histogram shows the distribution of the maximal number of MRs that affect each file in a day.

of summarization. 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 that 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 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 encounter here are at least as severe as those found in lesser sized and complicated developments. Thus we argue that our data has a good level of external validity and is generalizable to other developments of similar domains.

6 SUMMARY AND EVALUATION

6.1 Study Summary

This work represents initial empirical investigations to understand the nature of large scale parallel development. The data showed 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.
- 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 developers is 50% which while not concurrent with respect to the MR level 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.

6.2 Evaluation of Current Support

As we mentioned in the introduction to the data and analyses section, 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 4.4 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[9] 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 4.5 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, the supported strategy is what is left unsupported in these developments.

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 9 features have a very large tail distribution with a maximum number of files per feature show in Table 1 as 906. Synchronizing at the MR level is not a problem since most MRs affect only one file. However, each MR represents only a partial solution to a problem as represented by an IMR and is not a useful candidate for coherence and consistency. IMRs, on the other hand, each represent a specific problem in implementing a feature and have a mean value for the files affected of 4.3 per IMR. However, as we have noted elsewhere, the mean is not a satisfactory indication of the problem since the tail here is again a very substantial one having a maximum value of 388 files for at least on IMR.

Here again, further studies are needed to assess the appropriate level of parallelism that is useful in implementing suchs a synch and build strategy.

6.3 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 semantically. This phenomena requires us to look very closely at the files themselves via some program analysis tools.

Once we have this level of understanding and knowledge of interference, it will be interesting to see if there are any correlations between the associated quality data and programs where there are high degrees of parallel changes and/or interference.

ACKNOWLEDGEMENTS

The Code Decay Project [2] 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 wish to thank members of the project who have provided us with background information, insightful discussions, technical suggestions, and general support which led to this work.

REFERENCES

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

[2] Code decay home page. http://www.bell-labs.com/org/11359/projects/decay.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] David B. Leblang. Personal communication.

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

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

[13] 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.

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

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

[16] 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.

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

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