Position paper to be presented at the International Workshop on Principles of Software Evolution, Kyoto, Japan, April 1998

Challenges in Evolving a Large Scale Software Product

Harvey P. Siy
Software Production Research Department
Bell Laboratories
Naperville, IL 60566 USA
+1 630 224 6830
hpsiy@research.bell-labs.com

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

ABSTRACT

Evolving a large system presents a number of significant challenges. Not only is the developer concerned about how to fit in a new feature to a maze of existing features, he has to make sure his changes do not conflict with those being made in parallel by his colleagues. This is a minor problem in small projects with small organizations. However, as the project size scales up, so does the organization, and management of parallel tracks of development becomes a major concern. Moreover, increasing usage by customers with diverse needs pulls the evolving software into different directions, necessitating the evolution of multiple customized versions and compounding the already complex problem of evolving legacy systems.

We will examine one such legacy system, the Lucent Technologies 5ESS® switching system. First introduced in 1982, 5ESS was envisioned to support telecommunication needs well into the next century. Already one of the largest and most complex pieces of real time code in the world, the software to run the switch still continues to evolve with new features and in an increasing number of customized versions. In order to keep up with future evolution and maintain the growing base of customers, a combined procedural and technological solution was put in place. We will discuss this particular solution and its limitations.

1 INTRODUCTION

Any successful software product evolves in order to stay successful. Evolution involves fitting in new features into an existing maze of features. At any given time, multiple features are being implemented by multiple teams of developers. Not only is a developer concerned about unforeseen feature interactions between his team's feature and the existing base, he is concerned about undesirable interactions between his feature and those that are being made in parallel by his colleagues.

In many cases, a successful software product must evolve in more than one direction. To maintain the increasingly diverse (and potentially conflicting) needs of the growing customer base, multiple versions of the same product, each one tuned to specific customers, are spawned and evolved. This customization further complicates the job of a developer working on a new feature. Not only does he have to contend with his colleagues working on other features in parallel, he has to make sure his team's feature fits into one or more existing bases.

The effects of the fusion of parallel development and multiple versions worsens with scale. As a project grows, the bases become confusing mazes of features, constraining the ability to add new features. As the development organization grows, the level of awareness among developers go down, perhaps to the point where no single person has a clear overall picture of what is going on. Moreover, the amount of parallel evolution increases over time.

We will describe how one system is coping with this phenomenon. The 5ESS switching system is Lucent Technologies' flagship product, used in connecting local and long distance calls involving voice, data, and video communications. The software to run the switch is more than 10 million lines of code, divided into 50 subsystems, and involves more than 3,000 developers. The degree of parallel work going on is at an unprecedented level [4]. Moreover, the globalization of the telecommunications market has led to a diverse international customer set who have different requirements due to different telecommunication policies, maturity of telecommunications infrastructure, unwillingness to replace expensive legacy hardware, etc. With this increasing globalization, evolving multiple, customer-specific versions is increasingly becoming a concern.

To continuously evolve the system, a combined procedural and technological solution was put in place. We will discuss this particular solution and summarize with a discussion of issues that need to be addressed.

2 5ESS SOFTWARE 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 (defined below) applied

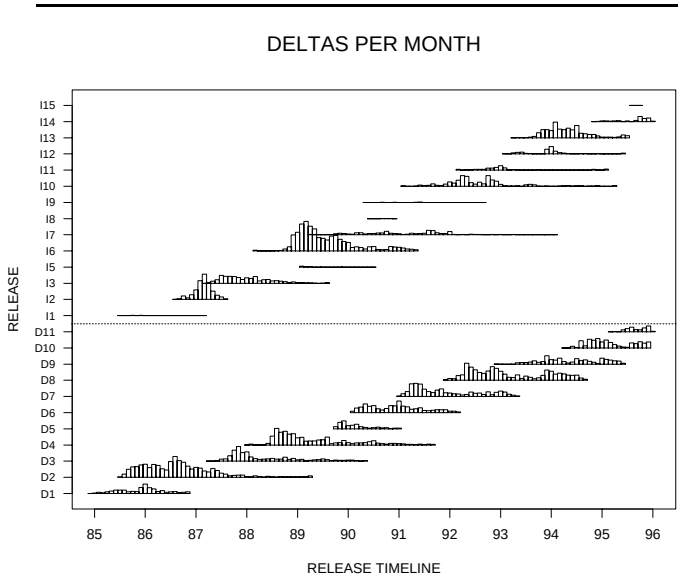

Figure 1: **Timeline of parallel releases.** Each histogram represents work being done for one release of the software. The top and bottom halves show releases for the international and domestic products, respectively.

every month to each release of one 5ESS subsystem. It gives a feel for the amount of parallel activity going on. 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 both development and maintenance (i.e., evolution) at any given time.

We will describe the software development process in 5ESS, focusing on the latter phases where most of the impact of parallel development and customizations are felt.

2.1 Making Changes

Change Management. Lucent Technologies uses a two-layered system for managing the evolution of 5ESS: a change management layer, ECMS [7], to initiate and track changes to the product, and a configuration management layer, SCCS [6], 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 MR. This includes 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.

Viewpathing. Viewpathing searches an ordered list of locations to find files. Each location in the viewpath is identified by a full pathname. Each pathname must be unique but the directories under them must have the same structure. The last location in the viewpath contains the official version of all files while the other locations may have just a subset of the files. In building a subsystem, the compiler tools searches through the viewpath in sequence from first to last for the files it needs. This idea allows developers to extract only the files they need to modify to a private path (which is then inserted to the front of the viewpath) while still being able to build the whole subsystem. Thus developers can make modifications, build and test their private copies.

MR Coding Process. 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. In fact, some files may be involved in more than 10 active MRs at one time [4], each with its own private copy of the file. 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.

A previous fault study [5] also showed that due to the amount of concurrent work going on, nearly 30% of all MRs for a given release are discarded, either because they were duplicate problems already reported by someone else, or they were unnecessary fixes requested by developers who misunderstood the system requirements.

2.2 Merging and Integration

Load Building. When the developers are done with their unit tests, their MRs are submitted to a central build team which performs the load building. A public build is attempted 3 weeks after the previous public build. A freeze date is announced to mark the start of the public build. Prior to the freeze date, the central build team will perform prebuilds of MRs that have already been submitted to it. In this way, some build errors can be caught before the actual public build. Depending on the number and severity of build errors, load building takes from 1 to 7 weeks.

Load Bringup. After the build has been successful, the load is distributed to the test labs and the test labs are set up with the proper environment. This turns out to be nontrivial, first of all, because any load this size simply takes a long time to duplicate and move around. In addition, the test lab environments evolve along with the particular releases. So depending on the release to be tested, the set up requires different sets of tools, operating system versions and manual hardware adjustments. The fault study previously mentioned [5] noted that almost 20% of faults in one release was due to problems with the test environment itself and/or setting it up.

System Testing. After the labs have been set up, regression tests are run on the different platforms. After regression testing has succeeded, the new features in the load are ready for the feature and system testers. Lab time is scheduled for the various tests that need to be performed. In many instances during testing, the testing time is insignificant compared to the set up time because the test fails almost as soon as it is run.

2.3 Managing Customization

Software Updates. According to Figure 1, every year or so, a new major release of the 5ESS system is made available. After the release date, each release goes into field support mode in which customer-found field problems are fixed. These fixes are patched onto the running system as *software updates*. The software update deployment mechanism can also be used to patch new features onto the running system. This uncoupled some features from following the mainstream development life cycle of the product release, allowing developers to deliver some features virtually on demand.

Streams. When there was only one version of a release, every time a new feature or fix was made, the software update had to be sent to every customer on the release, whether they needed it or not. This was potentially expensive for international customers. To reduce cost, improve turnaround time, and increase customer satisfaction, a multiple *stream* strategy was adopted, where a new feature or fix is only sent to the minimal set of customers who need or want the change. In the multiple-stream strategy, a stream is opened (if one does not already exist) for each customer requesting a new fix or feature. The software updates are then patched to the customers' streams. A customer may elect to upgrade to a newer release or stream, in which case he is taken off his current stream.

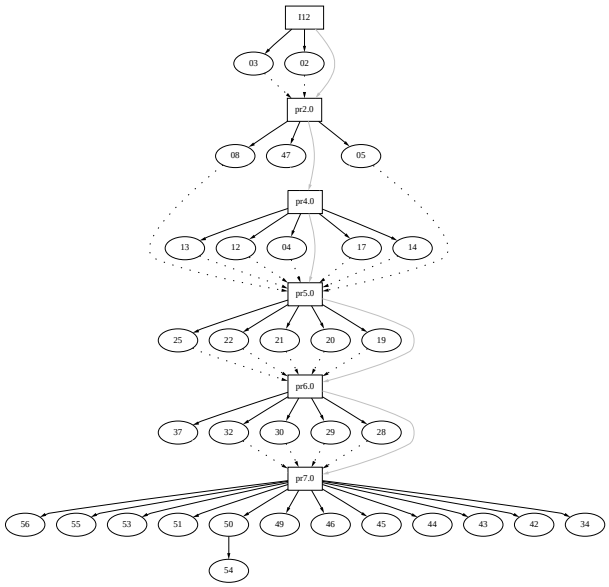

Figure 2: **Release I12 streams genealogy.** This genealogy tree illustrates the set of active streams for Release I12. The rectagular nodes are the PR loads. The oval nodes are the streams. The solid links connect base loads to the SU streams derived from them. The dotted links illustrate how some features in the streams propagate back to the PR loads.

Field Fixes. Sometimes a bug is found which impacts a set of releases. The change is made using special compiler directives. There are directives which specify the earliest applicable release, telling the compiler that the changes applies from that release onwards. Other directives tell the compiler when a change only applies up to a certain release.

2.4 Integration of Streams

Product Releases. Figure 2 shows the streams based off Release I12. Overall, there are about 100 streams being actively maintained by the development organization. At regular intervals, the load builders will incorporate all new features and fixes into a *product release* (PR) load. This becomes a new base upon which future features and fixes are made.

3 DISCUSSION

In order to continue evolving this and other large systems, several challenges must be met.

3.1 Merging Parallel Changes

The problems in parallel development arise when it is time to put them together.

Costs of Build Problems. Inevitably, problems will show up during the build stage due to late or missing MRs, name clashes, etc. These builds are expensive in terms of effort and time. It may take an effort-intensive search to track down the problem and track down the developer who introduced the problem. This delays the completion of the build process and disrupts tight lab schedules [8].

Coordination and Builds. The product build step is central to coordinating the separate threads of activity. Case studies have shown that the ability to perform frequent product integrations is proving to be a key factor in large successful projects [2]. It is easy to show that if the interval between builds is made longer, the probability of interactions increases geometrically as the degree of parallel development increases, quickly increasing the complexity of the conflict resolution process. One way to simplify this process is to keep the parallel activities low by shortening this interval. However, as the interval is shortened, the cost of performing the builds – a lot of which is attributed to the cost of setting up the labs – might grow prohibitively expensive. Thus a challenge is to find the optimal interval between builds to minimize cost and probability of interactions.

Detecting Interactions. Another option is to maintain a long interval between builds while being able to better handle the coordination process. So a primary area for tool support is in identifying potential interactions between changes in the different threads of activities and making sure that they are not in conflict with each other. Currently, the state of the practice for detecting interactions is to perform extensive system testing. The introduction of program slicing tools to detect interfering changes [3] may reduce the cost by catching interactions due to syntactic dependencies before testing. To detect interactions due to logical dependencies, a dynamic analysis approach for detecting changes in execution profiles such as dynamic path profiling [1] may be useful.

3.2 Change Propagation

Evolving multiple versions introduces several additional problems related to having to propagate code changes to multiple code bases.

Change Compatibility. Developers working on field fixes need to make sure their fixes work in all versions. This is a variation of the interaction detection problem where there are multiple code bases. There are bizarre cases where a fix may work for some versions but not for others due to some obscure logical dependency, thus requiring conditional fixes.

Distributing Fixes. With multiple versions, a mechanism is needed to propagate changes to the appropriate versions. Field problems found in earlier versions need to be propagated forward while problems found in later versions need to be propagated back to earlier ones.

Feature Migration. Another form of propagation happens when features in one stream are migrated to another stream or to the base code. This leads to arbitrary feature combinations, taxing the ability of the testing organization to ensure that the resulting systems remain reliable.

4 SUMMARY

In this paper, we have attempted to show that the evolution of a large scale real-time system is extremely complex in and of itself because of the amount of parallelism that goes on in evolving the system. This is compounded by market pressures for customer-specific features which may be folded into the base system as well as passed on to other customers. In addition, parts of the supporting environment have to evolve with the product itself.

This goes to show that the problem of software evolution is far richer than is generally recognized.

ACKNOWLEDGEMENTS

We thank Mary Denton and Brian Enke for answering all our numerous questions about software updates and the streams process.

REFERENCES

[1] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.

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

[3] 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.

[4] Dewayne Perry, Harvey Siy, and Lawrence Votta. Parallel changes in large scale software development: An observational case study. In *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[5] Dewayne E. Perry and Carol S. Stieg. Software faults in evolving a large, real-time system: a case study. In *4th European Software Engineering Conference – ESEC93*, pages 48–67, Sept. 1993. Invited keynote paper.

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

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

[8] Alexander L. Wolf and David S. Rosenblum. A study in software process data capture and analysis. In *Proceedings of the Second International Conference on the Software Process*, pages 115–124, Feb. 1993.