If Your Version Control System Could Talk ...

Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

Jung-Min Kim Adam A. Porter
Dept. of Computer Sciences
University of Maryland
aporter@cs.umd.edu

Harvey P. Siy
Bell Laboratories
Lucent Technologies
hpsiy@research.bell-labs.com

Abstract

Version control systems (VCSs) are used to store and reconstruct past versions of program source code. As a by-product they also capture a great deal of contextual information about each change. We will illustrate some ways to use this information to better understand a program's development history.

1 Introduction

There are many software-based metrics that one may use to assess the state of a software system. For example, the McCabe[7] and Halstead[4] software complexity metrics measure aspects of the structure of a static snapshot of source code to estimate its complexity. By measuring how these metrics change over time, the hope is that one can identify "decaying" components of a software system that, if restructured, may reduce the development effort needed to maintain and extend the system.

Such analyses depend on the ability to recreate snapshots of the software at different points in time. A version control system (VCS) tracks each change a developer makes to the system and, as a result, can recreate a consistent snapshot at any point in time. Examples of VCSs include RCS[10] and SCCS[9]. While this basic functionality of VCSs is essential for version control and measuring the evolution of metrics over time, there is much data in a VCS that is ignored when simply using it to extract snapshots of source code.

In this paper, we will examine some of the rich structure in version control data and show how analysis of version control data can illuminate the software development process in new ways. A VCS tags each change with a substantial amount of additional contextual information. Knowing what code was changed, when it was changed, who made the change, and so on, can yield valuable insights into what actually went on in the course of code development, sometimes better than developers' memories. VCS data is amenable to completely automated analysis, just as a snapshot of source code is amenable to metric analysis. Most software development organizations employ some form of VCS. Thus the methods we will outline here have wide applicability to many software projects.

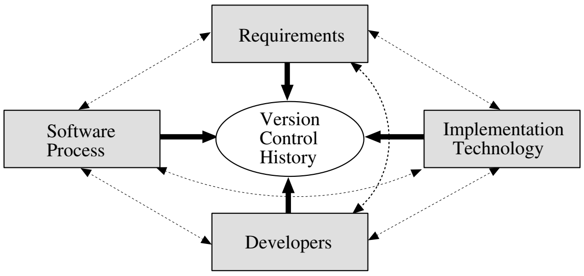

Figure 1: Version control history captures interactions between various aspects of software development.

Our framework is to use the data in a VCS (referred to as the version control history), in addition to the source code itself, to explore the relationships between various aspects of software development. For example, the requirements of the software, the implementation technology used, development process followed, and the organization of the developers have some effect on how the software evolved. Figure 1 illustrates the idea.

Each of these characteristics is a particular view of software development, with its own constraints and impact. Our thesis is that interactions between these views can impact the efficient production of software and that the version history data captured in a VCS provides a good starting point for exploring many of these relationships. In our particular example, we will examine some interesting relationships between the requirements of an optimizing compiler, the implementation technology of C++, and the desire to have developers work in parallel. We will examine these relationships using visualizations automatically generated from the VCS data.

2 Version Control Data

We present a very simplified description of some basic data collected by a VCS. Most VCSs operate over a set of files containing the text lines of source code. An *atomic* change to the program text is captured by recording the lines that were deleted and those that were added in order to make the change. This information is usually computed by a file differencing algorithm (such as Unix diff), which compares the previous version of a file with the current version. Each change has associated with it the file that was changed, the time the change was recorded by the VCS, and the name and login id of the programmer who made it, at a minimum.

In order to make a change to a software system, a developer may have to modify many program entities in many files. Most VCSs have the ability to track a group of related changes via a *modification record* (MR), which captures the fact that these changes were made for a specific purpose and are thus semantically related. An MR may have an English abstract associated with it that the developer provides, describing the purpose of the change. An MR also may have other data, such as it's opening time, closing time, and other status information (approved, rejected).

An MR is a set of atomic changes and every atomic change has some MR associated with it. Thus, every line of text in any source code snapshot has associated with it: time of change, developer name and login, MR and related data. By analyzing the source code to identify functions, classes or other program entities, we can then associate the VCS data with these entities as well as with source lines.

3 The Project Under Study

We present a brief overview of the project under study. More details may be found in Ladd and Ramming[6].

The 5ESS™ is Lucent Technologies' flagship local/toll switching system, containing an estimated 10 million lines of code in product and support tools. At the heart of the 5ESS software is a distributed relational database with information about hardware connections, software configuration, and customers. For the switch to function properly, this data must conform to certain integrity constraints. Some of these are logical constraints; for example, "call waiting and call forwarding/busy should never be active on the same line." Other constraints exist to document data design choices (redundancy, functional dependencies, distribution rules) that support efficient 5ESS operation and call processing.

PRL5[5] is a declarative SQL-like language, created to specify these data integrity constraints. PRL5 specifications are translated automatically into data audits and transaction guards in C, which is then compiled on multiple platforms. Due to the constantly changing integrity constraints to be provided to different communication service providers worldwide, compilation speed was crucial. The generated C code also had to be optimized to make as few disk accesses as possible.

A compiler for PRL5 called P5CC (PRL5 to C Compiler) was developed in C++. The compiler follows traditional compiler structure and includes lexing, parsing, semantic and type checking, optimization, and code generation phases. The lexing and parsing phases produce an abstract syntax tree (AST) over which the other phases operate.

For this compiler project, we have chosen to study the following views:

- *Requirements.* Build an optimizing compiler for a declarative SQL-like language
- *Developers.* Team of 6 developers who have to coordinate with each other
- *Implementation technology.* The C++ language
- *Software process.* Many, but not all components built in parallel.

4 Insights Into Compiler Development

The P5CC compiler is written in C++, using approximately 275 classes. Multiple inheritance is not used, so the class inheritance relationship is tree structured. Approximately, 120 of these classes are small stub classes generated via macros. We partitioned the classes into 5 basic groups (excluding miscellaneous support classes):

- *Top*: the top level classes. That is, the classes not directly related to any language entity, but used primarily as base classes;
- *SymTab*: symbol table classes representing a PRL5 language construct or type;
- *AST*: abstract syntax tree (AST) classes;
- *Optim*: classes that apply optimization transformations to the abstract syntax tree;
- *Quads*: code generation classes;

Each modification to the P5CC compiler is recorded in a VCS known as ECMS (the Extended Change Management System). Our analysis is based on 750 MR's. For each class, we computed how many MRs modified it. Figure 2 shows the inheritance hierarchy of the P5CC classes. The area of a node is proportional to the log of the number of MRs that modified the corresponding class. A node is only shown if there was an MR touching

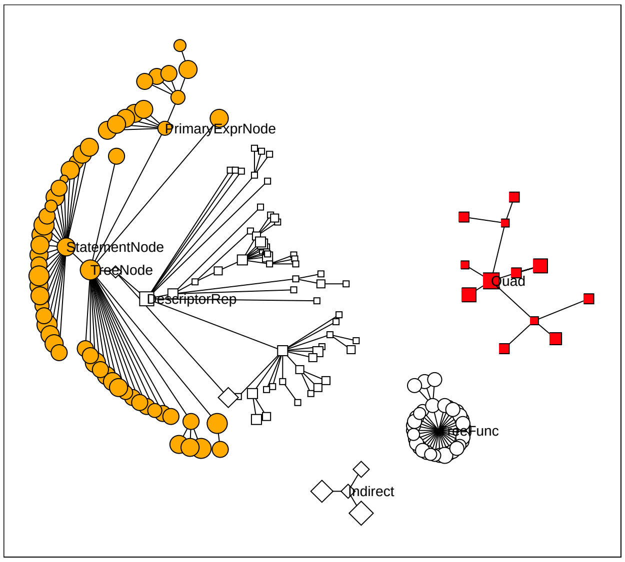

Figure 2: The class inheritance hierarchy of the P5CC compiler, where the size of a node is proportional to the log of the number of MRs that touch the class. The shape of the node represents its class group as follow: Top – diamonds, SymTab – unfilled square, AST – filled circle, Optim – unfilled circle, Quads – filled squares.

it; this is the reason that the number of *Quads* classes is so small–most of the classes are generated by macros and are never modified by hand.

It is clear from this figure that most of the change activity was related to the AST. Keep in mind that the AST code does much more than simply maintain a tree data structure. Member functions of the AST classes are responsible for many compilation phases, such as type checking, code generation, etc. In an imperative language such as C, one might encapsulate type checking in a module that takes an AST as input and traverses it to perform the type checking. However, in an object-oriented language such as C++, the function of type checking is more naturally expressed as a method of the AST classes.

In the P5CC compiler, the member functions of the AST classes are divided among files according to their functionality. For example, all the member functions for type checking can be found in a particular set of files, while member functions for code generation can be found in another file set. The left panel in Figure 3 makes this clear, using the SeeSoft code visualization [1]. Each rectangle represents a file from the compiler. Only files containing code from the AST classes are shown. Each class is given a level of gray from the gray scale spectrum on the left, and each line of a file is colored to show which class it belongs to. As is clear from the picture, each file contains code from many classes. Note, for example, the files containing a gray scale spectrum. These files group members functions from many different classes in the AST that constitute a particular phase of the compiler (such as type checking or code generation).

There is a very good reason to divide classes into files by functionality. Centralizing too much functionality in one set of classes can impede concurrent development. By splitting member functions of the same class across different files, different programmers can work on different aspects of the compiler. The right panel in Figure 3 shows which programmers worked on which files. This data is derived from ECMS: every MR is owned by exactly one programmer, so the code for that MR can be traced to an individual. It is clear that certain programmers had ownership of particular aspects of the compiler and AST classes, although most files have accumulated modifications by several programmers.

This partitioning illustrates the complex relationships among the views we are examining. For example, compiler literature[8] suggests that when prototyping a new programming language, it is useful to implement compiler phases as member functions of each AST class. To add new language constructs, one simply adds new classes to the set of AST classes. On the other hand, for a mature language, it is useful to implement compiler phases as modules that transform or operate over an abstract data type representing an AST. This makes it easier to add new optimizing passes to the compiler.

Since PRL5 was changing during the compiler's development, an OO approach would appear to make sense (the inheritance hierarchy acts as a built-in switch statement and virtual functions help avoid writing identical functions in all classes within a inheritance hierarchy). However, the developers' work assignments mimic'd the second approach. That is, each developer implemented a specific compiler phase for all AST classes. In the end, the compiler phases were incorporated into the member functions of the AST classes, but, during development, member functions with similar functionality were grouped together into separate files.

One goal of our research is to better understand the effect that development decisions, such as these, have on the evolution of software systems. In the next section we discuss some preliminary approaches for analyzing the relationship between development approaches and a system's change history.

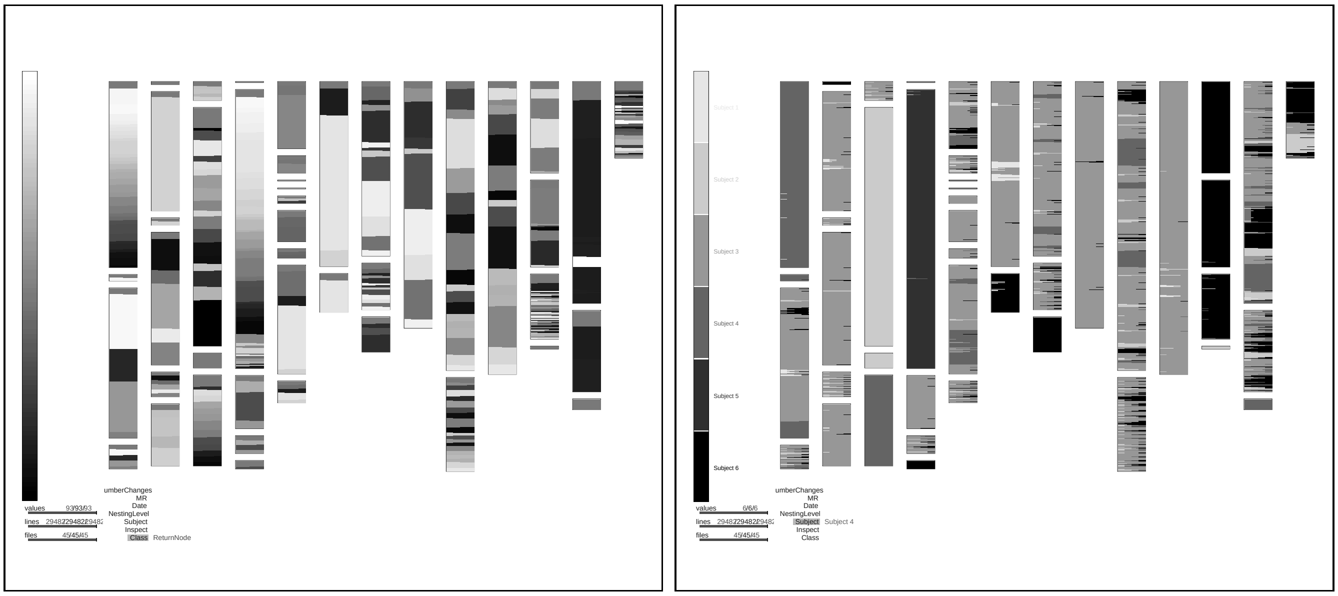

Figure 3: Two SeeSoft views of the files comprising the abstract syntax tree classes of the P5CC compiler. In the view to the left, each class maps to a level of gray in the gray scale spectrum on the left and each line of a file is colored with the class it belongs to. The view on the right shows the same set of files, where each line of a file is colored by programmer (there are six programmers).

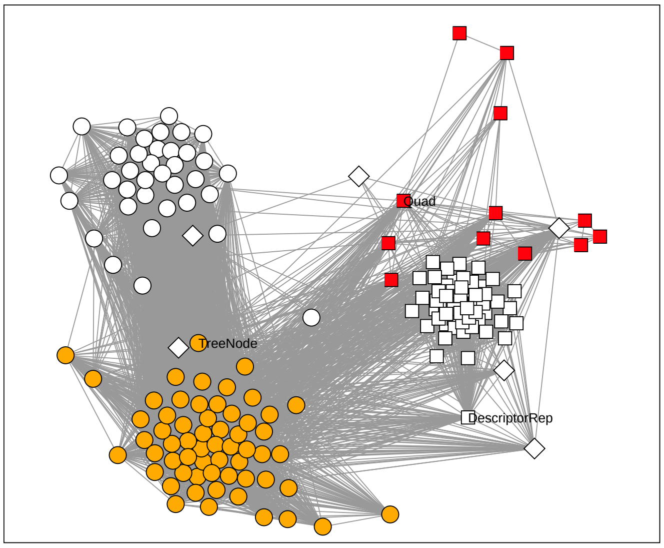

Figure 4: Cluster analysis of classes based on MR relationships. Each node representing a class is colored by the group of the class (same color and shape as in Figure 2). Nodes closer together are joined by links of higher probability. The clustering analysis has defined the five class groups.

5 Cluster Analysis of Classes by Modification

One way to help understand the effects of a development decision on system evolution is to examine the system's change history. In this example we describe initial explorations of how the class structure of this system encapsulated changes to it.

For each class C, let C_{mr} be the number of MRs that touch class C. If C and D are two unique classes, let CD_{mr} be the number of MRs that touch both classes C and D. CD_{mr} defines a relationship between the two classes, which translates to a link in a graph where the nodes are classes. A probability measure can also be associated with the link:

$$CD_{mr}/\sqrt{C_{mr}D_{mr}}$$

If the classes C and D are always modified together, the link probability will be 1, since $C_{mr} = D_{mr} = CD_{mr}$ in this case. However, if C and D are rarely modified together in comparison to the total number of modifications to C or D, then the link probability will be low.

We generated this data for the current version of the P5CC compiler and ran it through a graph clustering algorithm which places nodes connected by links of higher weight closer together. We used the probability measure described above for the link weighting. For details on this algorithm, see [3].

Figure 4 shows the resulting graph. Each node represents a class. The shape of a node denotes the group that the class belongs to, as in Figure 2. Note that the clustering algorithm has identified clusters of semantically related classes. In fact, it identified the five basic partitions described earlier! The cluster of round filled nodes on the lower left are the AST classes, while the cluster of white square nodes on the right are the *SymTab* classes. The cluster on the upper left is the *Optim* group. The group of filled square nodes above the symbol table cluster represents the *Quads* group. In this example, all links between classes of different groups had probabilities less than 40the same group had higher probability.

This figure shows that changes to the AST classes often involved other changes to the *Optim* and *SymTab* groups. However, as expected, there are very few links between the *Optim* and *SymTab* groups.

Based on this initial analysis, we are currently examining the types of changes that are made between classes within the same hierarchy. If they are mainly changes involving the same member function across different classes, then it argues that the class partitioning was not effective. In addition, we are examining similar views for file partitioning.

6 Summary

The thesis of this research is that version control systems contain significant amounts of data that could be exploited in the study of system evolution. We have illustrated some ways to use this information. In particular we have derived VCS-related metrics, like connection strength based on the probability that two classes are modified together. We used this metric to assess the effect of implementation decisions on the evolution of the resulting software.

Clearly, this work is in its initial stages. We are currently exploring several extensions to it.

- Time-series analysis. Most metric work is based on a single snapshot of a system. Little work has explored metrics to capture how structure changes over long periods of time.
- Improved analysis models. This work implicitly treats aspects of the change history as dependent variables and aspects of development history as independent variables. Our initial work has looked at only a few variables. For example, we have looked at number of changes per class as a dependent variable. In our ongoing work we will refine this variable, looking at the type of changes, change effort, and whether the change was actually approved or had to be reworked. We will also examine complex changes, i.e., those that involve multiple developers, multiple classes, and multiple functions.
- Examining development processes. We can also examine issues related to the development process. For example, are defects discovered during inspection different in nature than those discovered during testing? We can also look at the effect of development decisions on criteria such as development interval.
- Improved visualization techniques. The visualizations we show in this paper are obviously static. Since we are inherently interested in system behavior over time we expect that visualizations must improve to capture this. Some possibilities include Trellis displays[2] and animation.
- Static program analysis. The change history provides information about how different parts of a system are related. This information may be useful for automatic restructuring.

References

[1] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.

[2] William S. Cleveland. *Visualizing Data*. Hobart Press, 1993.

[3] Stephen G. Eick and Graham J. Wills. Navigating large networks with hierarchies. In *Visualization '93 Conference Proceedings*, pages 204–210, San Jose, California, Oct. 1993.

[4] Maurice H. Halstead. *Elements of Software Science*. Elsevier – North Holland, 1979.

[5] David A. Ladd and J. Christopher Ramming. Software research and switch software. In *International Conference on Communications Technology*, Beijing, China, 1992.

[6] David A. Ladd and J. Christopher Ramming. Two application languages in software production. In *USENIX Symposium on Very-High-Level Languages*, Oct. 1994.

[7] Thomas J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, Dec. 1976.

[8] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of 1997 ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, Jan. 1997.

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

[10] 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, Sept. 1982.