Analyzing Source Code in Source Control Repositories

Position Paper

Harvey P. Siy
Dept. of Computer Science
University of Nebraska at Omaha
hsiy@mail.unomaha.edu

Dewayne E. Perry
Dept. of Electrical and Computer Engineering
University of Texas at Austin
perry@ece.utexas.edu

ABSTRACT

Empirical studies based on version control repositories often make use of data *about* deltas, ignoring the source code fragments comprising the body of the deltas. Analysis techniques that mine the information in the delta body can substantially augment the data available from source control repositories. We sketch 2 approaches to analyzing the delta body, applying the analyses to the identification of interfering changes.

1. INTRODUCTION

Source control repositories store information that is used to recreate any version of a given source code artifact. Many empirical case studies rely on information about the source code changes, e.g., size of change, duration and number of related changes, developer who made the change, etc. These data, often obtained in combination with higher level change management repositories and project databases, have been indispensable in assessing maintainability of legacy code, identifying defect-prone components, evaluating the impact of process and technological changes, and studying various other evolution phenomena.

In most studies, the source code itself is generally ignored. As a result, there is a limit to how much information can be deduced from such analysis because there is little knowledge of what was actually being changed. To get to this knowledge, we need to analyze the source code and augment the change data with semantic information mined from the fragments that were actually inserted, deleted or modified.

Part of the reason that there are few results in this direction is the difficulty of working with source code. Program analysis tools work on versions (snapshots) of code. For long-lived sources, there can easily be hundreds, perhaps thousands of versions for every source file. We need analysis tools that scale up not only in terms of the size of the version being analyzed but also in terms of the number of versions being analyzed. Also, most program analysis tools, especially for identifying data dependencies (slices), require that the source should compile. We need analysis techniques that relax this requirement.

The first need is addressed by developing code analysis techniques that work directly with the native source control repository format. Typically, systems like SCCS and CVS store their delta information together with the code fragments in the same file. It is therefore sensible to have tools that directly "parse" those files and do not require recreating individual versions. This also allows us to take advantage of the relative similarities between any pair of successive versions.

The second need stems from the recognition that we often do not need (not immediately) the level of precision provided by standard tools that perform sound static analysis of the code. We can live with some uncertainty as a result of using approximate analysis techniques.

In this paper, we outline some approaches for efficiently performing approximate analysis of source code fragments. In the process, we also present some new visualization techniques to support the analysis.

2. MOTIVATIONS

The impetus for this work arose from the need to identify and measure the amount of semantic interferences between changes to the code that happen in parallel. [2] To detect semantic interference between two deltas, the straightforward approach is to extract a version of the source where both deltas are applied, perform program slicing, and verify that there is a slice which includes both deltas. This is a tedious process as the appropriate version of the source has to be identified first. Furthermore, program analysis tools require the code to compile in order to build their internal representations. (For example, see [1].) This has several problems when mining large source control repositories:

- There is no guarantee of the syntactic correctness of any given version.
- Language definitions change over time. What was once syntactically correct may no longer be true today.
- Even if the version is syntactically correct, it is often missing necessary header files. Much effort is then spent trying to figure out the correct versions of the header files to use.

To address these problems, tools are proposed in this paper that operate on the version history files themselves. These tools trade the precision of using program analysis tools in exchange for the ability to analyze many versions of code.

Our approach does not exclude the use of program analysis tools altogether. Once versions of the code have been isolated and identified as interesting, we can use these tools to conduct more precise analyses.

3. REPOSITORY FILE FORMATS

We examine briefly how several source control repositories store version history. We will describe how delta texts are stored in SCCS. A comparison is then made to the RCS file format used by CVS.

3.1 SCCS file format

SCCS stores all text changes inline. [3] Whenever a new version is checked in, it is compared against the most recent saved version. At each location where a set of lines had been added, the added text is inserted into the SCCS file, bracketed between a pair of control lines `^AI` and `^AE`. (`^A` stands for the Control-A character.) At each location where a set of lines had been deleted, the deleted text in the SCCS is bracketed by another pair of control lines `^AD` and `^AE`. A set of lines that had been modified is treated as a deletion and an insertion. An example follows:

```
...
^AI 1
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

^AD 2
    for (i=0; i<argc-1; i++) {
^AE 2
^AI 2
    printf("The number of arguments is %d\n", argc);
^AD 3
    for (i=0; i<argc; i++) {
^AE 3
^AI 3
    for (i=1; i<argc; i++) {
^AE 3
^AE 2
        printf("arg = %s\n", argv[i]);
    }
    return(0);
}
^AE 1
```

The number at the end of each control line is the number of the delta that inserted or deleted the enclosed line.

3.2 RCS file format

RCS adopts a different approach to store text changes.[1] It stores the most recent version of the code and stores a log of the deltas in the reverse direction. [4] Whenever a new version is checked in, its entire text replaces the text of the most recent version. It is then compared against the version it just replaced. At each location where a set of lines had been added in the new version, a log entry is appended to the RCS file specifying a *deletion* of the inserted lines, meaning that the lines that were added in the new version should be deleted if recovering the previous version. Analogously, if a set of lines had been deleted in the new version, a log entry is appended to the RCS file specifying an *insertion* of the deleted lines, followed by the text of the lines that were deleted. Applying the same changes to the example in Section 3.1:

```
...
1.3
...
@#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf("The number of arguments is %d\n", argc);
    for (i=1; i<argc; i++) {
        printf("arg = %s\n", argv[i]);
    }
    return(0);
}
@

1.2
...
@d8 1
a8 1
    for (i=0; i<argc; i++) {
@

1.1
...
@d7 2
a8 1
    for (i=0; i<argc-1; i++) {
@
...
```

3.3 A uniform data representation

Rather than dealing with multiple formats, an alternative is to preprocess the files into a uniform data representation from which subsequent analysis can be carried out. The delta body can be thought of conceptually as an array in which every line that was ever inserted has an entry. Each entry includes:

1. *dinsert* – the delta that inserted that line
2. *ddelete* – the delta that deleted that line
3. *dtext* – the line of text inserted

An important constraint is that the line ordering must preserve the line order for all versions. In the preceding example, the `for` statement versions must be stored such that they occur before the body of the `for` statement. This constraint makes it relatively simple to compose together the lines for a given version: a line is included if the delta number of the version is between *dinsert* and *ddelete*.

Construction of this representation from SCCS is straightforward because SCCS deltas are stored inline. Constructing this from RCS is more complicated, however an algorithm based on the *piece table* concept [4] can yield a relatively efficient implementation.

Figure 1 illustrates this representation for an actual SCCS file. Every line of code ever inserted is represented by a horizontal line. Each line is plotted starting at the time of the delta that inserted it up to the time of the delta that deleted it. The color coding in this figure reflects the degree of indentation.

Figure 2 is a visualization of the version history of Figure 1. For each version, the lines that were included are identified and gaps resulting from the excluded lines are removed. This view enables us to see the growth of each version. Each version is plotted along the x-axis according to its delta creation time. The color coding again reflects the

[1] In this paper, we discuss only the case for the main trunk. Branch deltas are not covered.

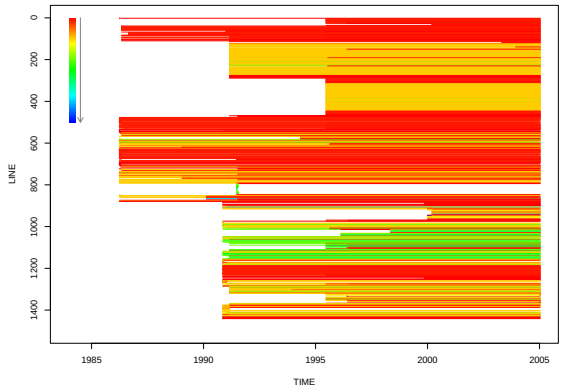

Figure 1: Visualization of deltas from an SCCS file. Each horizontal line represents the lifetime of a line of text. The color coding reflects the degree of indentation, ranging from red (little indentation) to blue (deep indentation).

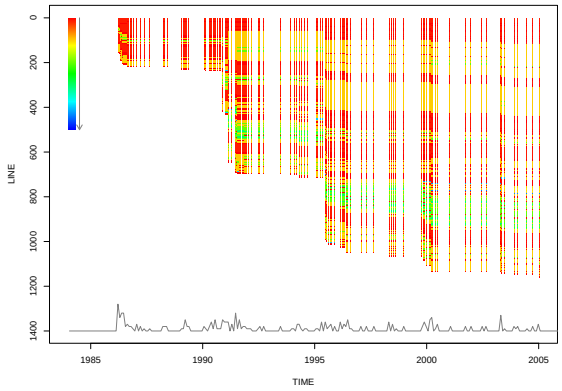

Figure 2: Visualization of deltas from an SCCS file. Each vertical line represents one version. The x-axis marks when the delta for that version was created. The line graph at the bottom of the plot indicates the frequency of monthly delta activities.

degree of indentation and helps to give a sense of continuity or discontinuity between successive versions, especially if dramatic changes are occurring. Since the deltas are plotted on a timeline and deltas tend to occur in bunches, it is not easy to visually pick out periods of high delta activity. Therefore we included the line graph at the bottom of the plot to indicate the frequency of monthly delta activities.

This visualization style is borrowed from CVSscan [5]. CVSscan visualized every version of a file as a column with each row color-coded according to some attribute of the particular line of code, such as indentation. The x-axis on CVSscan is the delta number instead of the delta creation time.

4. SOURCE CODE ANALYSES

We now discuss how we might approach the problem of detecting semantic interference.

4.1 Text-based Analysis

The first approach assumes that we have already performed a data dependency analysis on the most recent version of the source and we want to know if a pair of dependent statements had, in their past, been changed in parallel.

We identify the set of deltas that modified a particular line by tracing its version history using text-based analysis. The difficulty with tracing is that source control repositories do not record modifications *per se*, but as insertions and deletions. Since a line may be modified several times during its life, we define a heuristic for identifying the line in its earlier incarnations: as we trace back the history of a line, we look at when it was inserted, and from there, whether a "similar" line was deleted, and trace that line further back.

Since source control repositories operate on lines, this task involves a textual comparison of lines. A line-oriented differencing tool can be integrated into the algorithm. The general algorithm is outlined below. From a given line l which is an index into the data representation described in Section 3.3:

```
Q.enqueue((l, 1))
DeltaSet.init(LINES[l].dinsert)
while ¬Q.empty() do
    (l, sim) ← Q.dequeue()
    for { dl | within RANGE of l }
        if LINES[l].dinsert = LINES[dl].ddelete
          then
                newsim ←
                diff(LINES[l].dtext, LINES[dl].dtext)
                if newsim > THRESHOLD
                  then
                        Q.enqueue((dl, sim * newsim))
                        DeltaSet.add(LINES[dl].dinsert)
```

sim is defined as a similarity measure between 0 and 1. As the trace proceeds further back, the absolute similarity value is passed along in the queue. `diff` is defined to be a text differencing function similar to Perl's *Algorithm::diff* module, except that it returns a similarity value instead of the actual differences. *DeltaSet* set is the candidate set of deltas involved.

It is possible that more than one candidate line can be found. Based on the algorithm above, all candidate lines will be traced. Alternately, an interactive approach can be taken where the user chooses which one to follow.

The delta visualization from the previous section can be used to trace the history of a line or a set of lines. The color fades as the absolute similarity with the original line decreases.

4.2 Syntactic Analysis

The second approach involves syntactic analysis of the delta text.[2] First, the delta texts are parsed as explained below (Section 4.2.1). The goal is to annotate each delta with *input* and *output* variables. Input variables to a delta are those referenced within that delta. Output variables are those whose values may have been changed within that delta. There will also be variables we will call *indeterminate* because we cannot tell if they are inputs or outputs.

[2]For this discussion, we will assume that the programming language is C, although much of it can be adapted to any language.

Second, time-varying call dependencies are extracted from deltas involved in function calls and function heads. These are annotated with the versions where they hold. Indeterminate variables are resolved as input or output variables to the extent possible. The function each delta belongs to is also identified.

We now have the information needed to check if a given pair of deltas might be interfering. If the deltas are in the same function, we make a quick comparison to verify if the input variable in one is an output variable in another delta. If the deltas are in different functions related by an applicable call dependency, the formal parameters must be translated to the actual parameters first.

4.2.1 Parsing the text of a delta

The general strategy is to approach this as one would when cursorily reading a source file. We make certain assumptions:

1. Any given version has only minor or no syntax errors. This assumption allows us to parse the program statements using a loose grammar that accepts a superset of the language, as outlined below. While the code is not required to be syntactically correct, we expect more meaningful results when it is closer to being syntactically correct.
2. Identifiers whose definitions are not present are assumed to be declared in some other files. This assumption eliminates the need for finding other files to understand the behavior of one source file. There is no need for preprocessors as well.
3. Coding styles are generally followed. For example, identifiers that are in all capital letters are assumed to be macro definitions. Also, multi-line comments (enclosed by `/* */`) start every new line with a `*`. This lets us easily filter out comments and compiler directives.

The text being inserted or deleted may contain one of the following:

- elementary statement,
- variable, function or type declaration,
- function head,
- decision or loop head (`if`, `while`, `switch`, `do`),
- compound variable or type declaration head,
- compound statement or data delimeter ({ or }),
- compiler directive, comment or blank line

Since the inserted or deleted line of code may only constitute a partial statement, the first step is to reconstruct the whole statement. With C or C-like languages, an approximate statement can be reconstructed by searching downward until a line with ';' or '}' is encountered, and upward until a line with ';' or '{' is encountered.

Once a complete statement text has been reconstructed, the text can be tokenized and then parsed using a grammar oriented towards recognizing statements. Some elements of this grammar include:

⟨*Stmt*⟩ ::= ⟨*CStmt*⟩ | ⟨*Head*⟩ | ⟨*Tail*⟩
⟨*Head*⟩ ::= ⟨*FuncHead*⟩ | ⟨*StmtHead*⟩ | '`do`' | ⟨*DeclHead*⟩
⟨*StmtHead*⟩ ::= ('`while`' | '`if`' | '`switch`' | '`for`') '`(`' ⟨*Expr*⟩ '`)`'
⟨*FuncHead*⟩ ::= ⟨*Type*⟩ ⟨*Identifier*⟩ '`(`' ⟨*ParamList*⟩ '`)`'
⟨*ParamList*⟩ ::= ϵ | ⟨*VarDecl*⟩ | ⟨*ParamList*⟩ '`,`' ⟨*VarDecl*⟩
⟨*Tail*⟩ ::= '`while`' '`(`' ⟨*Expr*⟩ '`)`' | '`else`'
⟨*CStmt*⟩ ::= ⟨*Asgn*⟩ | ⟨*VarDecl*⟩ | ⟨*Expr*⟩ | ⟨*Jump*⟩
⟨*VarDecl*⟩ ::= ⟨*Type*⟩ ⟨*IdentifierList*⟩
⟨*IdentifierList*⟩ ::= ⟨*Identifier*⟩ | ⟨*Identifier*⟩ '`,`' ⟨*IdentifierList*⟩
⟨*Asgn*⟩ ::= ⟨*LVal*⟩ '`=`' ⟨*Expr*⟩
⟨*LVal*⟩ ::= ⟨*Identifier*⟩ | ⟨*Expr*⟩
⟨*Expr*⟩ ::= (⟨*Identifier*⟩ | ⟨*Operator*⟩)+
⟨*Jump*⟩ ::= '`return`' [⟨*Expr*⟩] | '`break`' | '`continue`'

Assuming that the original text can be succesfully compiled by a C compiler, each statement can be successfully parsed by this loose grammar. From the mini-syntax tree created, variables are extracted and classified as input or output. There will also be many indeterminate variables, especially arguments to function calls.

For deltas with multiple statements, the input and output sets of each statement are unioned together.

5. DISCUSSIONS

We have briefly sketched a pair of approaches for approximate analysis of the source text within a delta. Development of prototypes is underway and will be applied to the identification of semantic interference in a large source control repository for an industrial system. A key question to answer is effectiveness, whether the approaches can be tuned such that the proportion of false positives will not be overwhelming.

We also note that these techniques provide information for other kinds of tasks. Tracing the history of a crucial function can be used to add more weight to all deltas that affected that function. Tracing the history of a line of code can be used to discover change rationale.

Likewise, the syntactic information obtained can be used to assess the relative importance of a delta in terms of the number of deltas related to it. This can be an indicator of the potential for problems when a change is added on top of this one, as well as the effort required to get the change right.

Finally, in our discussion of representing and storing the delta information, we have assumed that all deltas are applied sequentially. This is not necessarily the case in many development processes which track multiple versions from the same source file. This implies changes to the way versions are calculated since not every sequentially defined delta is applied.

6. REFERENCES

[1] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the International Workshop on Mining Software Repositories (MSR'2005)*, St. Louis, Missouri, May 2005.

[2] D. Perry, H. Siy, and L. Votta. Parallel changes in large scale software development: An observational case study. *ACM Trans. on Software Engineering and Methodology*, 10(3):308–337, July 2001.

[3] M. J. Rochkind. The Source Code Control System. *IEEE Trans. on Software Engineering*, 1(4):364–369, Dec. 1975.

[4] W. F. Tichy. RCS–a system for version control. *Software Practice & Experience*, 15(7):637–654, July 1985.

[5] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: Visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, St. Louis, Missouri, May 2005.