Understanding Source Code Evolution Using Abstract Syntax Tree Matching

Iulian Neamtiu
neamtiu@cs.umd.edu

Jeffrey S. Foster
jfoster@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Department of Computer Science
University of Maryland at College Park

ABSTRACT

Mining software repositories at the source code level can provide a greater understanding of how software evolves. We present a tool for quickly comparing the source code of different versions of a C program. The approach is based on partial abstract syntax tree matching, and can track simple changes to global variables, types and functions. These changes can characterize aspects of software evolution useful for answering higher level questions. In particular, we consider how they could be used to inform the design of a dynamic software updating system. We report results based on measurements of various versions of popular open source programs, including BIND, OpenSSH, Apache, Vsftpd and the Linux kernel.

Categories and Subject Descriptors

F.3.2 [**Logics And Meanings Of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Languages, Measurement

Keywords

Source code analysis, abstract syntax trees, software evolution

1. INTRODUCTION

Understanding how software evolves over time can improve our ability to build and maintain it. Source code repositories contain rich historical information, but we lack effective tools to mine repositories for key facts and statistics that paint a clear image of the software evolution process.

Our interest in characterizing software evolution is motivated by two problems. First, we are interested in *dynamic software updating* (DSU), a technique for fixing bugs or adding features in running programs without halting service [4]. DSU can be tricky for programs whose types change, so understanding how the type structure of real programs changes over time can be invaluable for weighing the merits of DSU implementation choices. Second, we are interested in a kind of "release digest" for explaining changes in a software release: what functions or variables have changed, where the hot spots are, whether or not the changes affect certain components, etc. Typical release notes can be too high level for developers, and output from `diff` can be too low level.

To answer these and other software evolution questions, we have developed a tool that can quickly tabulate and summarize simple changes to successive versions of C programs by partially matching their abstract syntax trees. The tool identifies the changes, additions, and deletions of global variables, types, and functions, and uses this information to report a variety of statistics.

Our approach is based on the observation that for C programs, function names are relatively stable over time. We analyze the bodies of functions of the same name and match their abstract syntax trees structurally. During this process, we compute a bijection between type and variable names in the two program versions. We then use this information to determine what changes have been made to the code. This approach allows us to report a name or type change as single difference, even if it results in multiple changes to the source code. For example, changing a variable name from `x` to `y` would cause a tool like `diff` to report all lines that formerly referred to `x` as changed (since they would now refer to `y`), even if they are structurally the same. Our system avoids this problem.

We have used our tool to study the evolution history of a variety of popular open source programs, including Apache, OpenSSH, Vsftpd, Bind, and the Linux kernel. This study has revealed trends that we have used to inform our design for DSU. In particular, we observed that function and global variable additions are far more frequent than deletions; the rates of addition and deletion vary from program to program. We also found that function bodies change quite frequently over time, but function prototypes change only rarely. Finally, type definitions (like `struct` and `union` declarations) change infrequently, and often in simple ways.

2. APPROACH

Figure 1 provides an overview of our tool. We begin by parsing the two program versions to produce abstract syntax trees (ASTs), which we traverse in parallel to collect type and name mappings. With the mappings at hand, we then detect and collect changes to report to the user, either

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR '05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...$5.00.

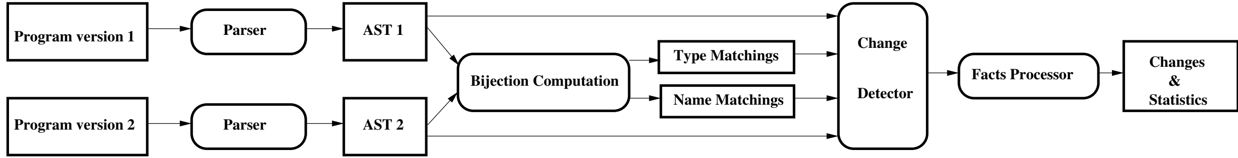

Figure 1: High level view of AST matching

```c
typedef int sz_t;
int count;
struct foo {
  int i;
  float f;
  char c;
};
int baz(int a, int b) {
  struct foo sf;
  sz_t c = 2;
  sf.i = a + b + c;
  count++;
}
```

```c
int counter;
typedef int size_t;
struct bar {
  int i;
  float f;
  char c;
};
int baz(int d, int e) {
  struct bar sb;
  size_t g = 2;
  sb.i = d + e + g;
  counter++;
}
void biff(void) { }
```

Figure 2: Two successive program versions

```
procedure GenerateMaps(Version1, Version2)
 F1 ← set of all functions in Version 1
 F2 ← set of all functions in Version 2
 global TypeMap ← ∅
 global GlobalNameMap ← ∅
 for each function f ∈ F1 ∩ F2
    do { AST1 ← AST of f in Version 1
         AST2 ← AST of f in Version 2
         Match_Ast(AST1, AST2)

procedure Match_Ast(AST1, AST2)
 local LocalNameMap ← ∅
 for each (node1, node2) ∈ (AST1, AST2)
    do { if (node1, node2) = (t1 x1, t2 x2)   //declaration
           then { TypeMap ← TypeMap ∪ {t1 ↔ t2}
                  LocalNameMap ← LocalNameMap ∪ {x1 ↔ x2}
         else if (node1, node2) = (y1 := e1 op e1', y2 := e2 op e2')
           then { Match_Ast(e1, e2)
                  Match_Ast(e1', e2')
                  if isLocal(y1) and isLocal(y2) then
                    LocalNameMap ← LocalNameMap ∪ {y1 ↔ y2}
                  else if isGlobal(y1) and isGlobal(y2) then
                    GlobalNameMap ← GlobalNameMap ∪ {y1 ↔ y2}
         else if ...
         else break
```

Figure 3: Map Generation Algorithm

directly or in summary form. In this section, we describe the matching algorithm, illustrate how changes are detected and reported, and describe our implementation and its performance.

2.1 AST Matching

Figure 2 presents an example of two successive versions of a program. Assuming the example on the left is the initial version, our tool discovers that the body of `baz` is unchanged—which is what we would like, because even though every line has been syntactically modified, the function in fact is structurally the same, and produces the same output. Our tool also determines that the type `sz_t` has been renamed `size_t`, the global variable `count` has been renamed `counter`, the structure `foo` has been renamed `bar`, and the function `biff()` has been added. Notice that if we had done a line-oriented `diff` instead, nearly all the lines in the program would have been marked as changed, providing little useful information.

To report these results, the tool must find a *bijection* between the old and new names in the program, even though functions and type declarations have been reordered and modified. To do this, the tool begins by finding function names that are common between program versions; our assumption is that function names do not change very often. The tool then uses partial matching of function bodies to determine name maps between old and new versions, and finally tries to find *bijections* i.e., one-to-one, onto submaps of the name maps.

We traverse the ASTs of the function bodies of the old and new versions in parallel, adding entries to a *LocalNameMap* and *GlobalNameMap* to form mappings between local variable names and global variable names, respectively. Two variables are considered equal if we encounter them in the same syntactic position in the two function bodies. For example, in Figure 2, parallel traversal of the two versions of `baz` results in the *LocalNameMap*

$$\texttt{a} \leftrightarrow \texttt{d}, \texttt{b} \leftrightarrow \texttt{e}, \texttt{sf} \leftrightarrow \texttt{sb}, \texttt{c} \leftrightarrow \texttt{g}$$

and a *GlobalNameMap* with `count` ↔ `counter`. Similarly, we form a *TypeMap* between named types (`typedef`s and aggregates) that are used in the same syntactic positions in the two function bodies. For example, in Figure 2, the name map pair `sb` ↔ `sf` will introduce a type map pair `struct foo` ↔ `struct bar`.

We define a *renaming* to be a name or type pair $j_1 \rightarrow j_2$ where $j_1 \leftrightarrow j_2$ exists in the bijection, j_1 does not exist in the new version, and j_2 does not exist in the old version. Based on this definition, our tool will report `count` → `counter` and `struct foo` → `struct bar` as renamings, rather than additions and deletions. This approach ensures that consistent renamings are not presented as changes, and that type changes are decoupled from value changes, which helps us better understand how types and values evolve.

Figure 3 gives pseudocode for our algorithm. We accumulate global maps *TypeMap* and *GlobalNameMap*, as well as a *LocalNameMap* per function body. We invoke the routine Match_Ast on each function common to the two versions. When we encounter a node with a declaration $t_1\ x_1$ (a declaration of variable x_1 with type t_1) in one AST and $t_2\ x_2$ in the other AST, we require $x_1 \leftrightarrow x_2$ and $t_1 \leftrightarrow t_2$. Similarly, when matching statements, for variables y_1 and y_2 occurring in the same syntactic position we add type pairs in the *TypeMap*, as well as name pairs into *LocalNameMap* or *GlobalNameMap*, depending on the storage class of y_1 and y_2. *LocalNameMap* will help us detect functions which are identical up to a renaming of local and formal variables, and *GlobalNameMap* is used to detect renamings for global variables and functions. As long as the ASTs have the same shape, we keep adding pairs to maps. If we encounter an AST mismatch (the **break** statement on the last line of the algorithm), we stop the matching process for that function and use the maps generated from the portion of the tree that did match.

```
------- Global Variables  ----------
Version1 :                         1
Version2 :                         1
renamed :                          1

------- Functions  -----------------
Version1 :                         1
Version2 :                         2
added :                            1
locals/formals name changes :      4

------- Structs/Unions  ------------
Version1 :                         1
Version2 :                         1
renamed :                          1

------- Typedefs  -----------------
Version1 :                         1
Version2 :                         1
renamed :                          1
```

Figure 4: Summary output produced for the code in Figure 2

The problem with this algorithm is that having insufficient name or type pairs could lead to renamings being reported as additions/deletions. The two reasons why we might miss pairs are partial matching of functions and function renamings. As mentioned previously, we stop adding pairs to maps when we detect an AST mismatch, so when lots of functions change their bodies, we miss name and type pairs. This could be mitigated by refining our AST comparison to recover from a mismatch and continue matching after detecting an AST change. Because renamings are detected in the last phase of the process, functions that are renamed don't have their ASTs matched, another reason for missing pairs. In order to avoid this problem, the bijection computation and function body matching would have to be iterated until a fixpoint is reached.

In practice, however, we found the approach to be reliable. For the case studies in section 3, we have manually inspected the tool output and the source code for renamings that are improperly reported as additions and deletions due to lack of constraints. We found that a small percentage (less than 3% in all cases) of the reported deletions were actually renamings. The only exception was an early version of Apache (versions 1.2.6-1.3.0) which had significantly more renamings, with as many as 30% of the reported deletions as spurious.

2.2 Change Detection and Reporting

With the name and type bijections in hand, the tool visits the functions, global variables, and types in the two programs to detect changes and collect statistics. We categorize each difference that we report either as an addition, deletion, or change.

We report any function names present in one file and not the other as an addition, deletion, or renaming as appropriate. For functions in both files, we report that there is a change in the function body if there is a difference beyond the renamings that are represented in our name and type bijections. This can be used as an indication that the semantics of the function has changed, although this is a conservative assumption (i.e., semantics preserving transformations such as code motion are flagged as changes). In our experience, whenever the tool detects an AST mismatch, manual inspection has confirmed that the function seman-

```
/ : 111
     include/ : 109
          linux/ : 104
               fs.h : 4
               ide.h : 88
               reiserfs_fs_sb.h : 1
               reiserfs_fs_i.h : 2
               sched.h : 1
               wireless.h : 1
               hdreg.h : 7
          net/ : 2
               tcp.h : 1
               sock.h : 1
          asm-i386/ : 3
               io_apic.h : 3
     drivers/ : 1
          char/ : 1
               agp/ : 1
                    agp.h : 1
     net/ : 1
          ipv4/ : 1
               ip_fragment.c : 1
```

Figure 5: Density tree for struct/union field additions (Linux 2.4.20 vs. 2.4.21)

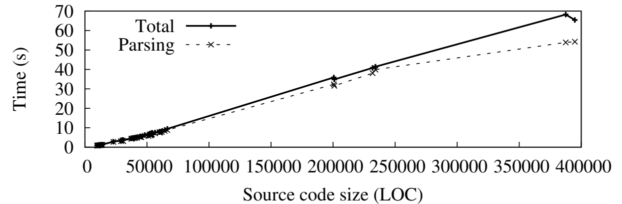

Figure 6: Performance

tics has indeed changed.

We similarly report additions, deletions and renamings of global variables, and changes in global variable types and static initializers.

For types we perform a deep structural isomorphism check, using the type bijection to identify which types should be equal. We report additions, deletions, or changes in fields for aggregate types; additions, deletions, or changes to base types for typedefs; and additions, deletions, or changes in item values for enums.

Our tool can be configured to either report this detailed information or to produce a summary. For the example in Figure 2, the summary output is presented in Figure 4. In each category, `Version1` represents the total number of items in the old program, and `Version2` in the new program. For brevity we have omitted all statistics whose value was 0 e.g., enums, etc.

Our tool can also present summary information in the form of a *density tree*, which shows how changes are distributed in a project. Figure 5 shows the density tree for the number of struct and union fields that were added between Linux versions 2.4.20 and 2.4.21. In this diagram, changes reported at the leaf nodes (source files) are propagated up the branches, making clusters of changes easy to visualize. In this example, the `include/linux/` directory and the `include/linux/ide.h` header file have a high density of changes.

2.3 Implementation

Our tool is constructed using CIL, an OCaml framework

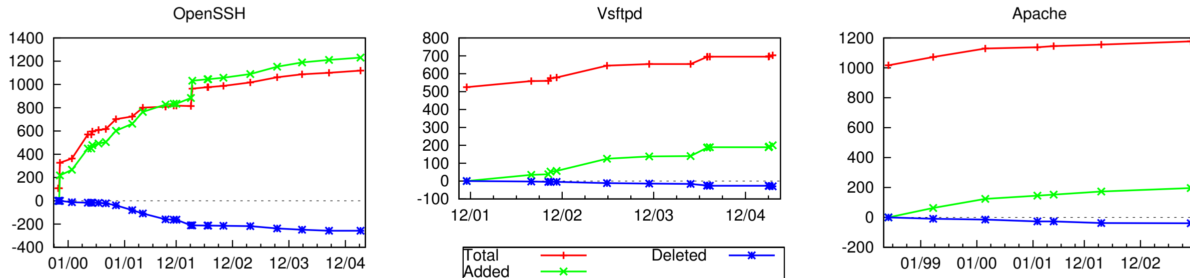

Figure 7: Function and global variable additions and deletions

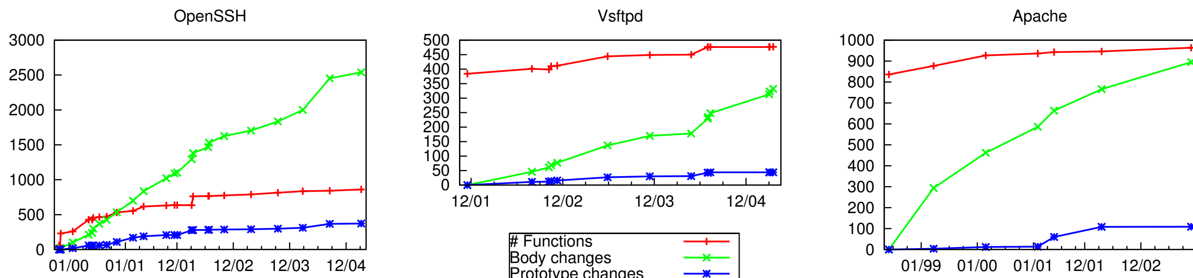

Figure 8: Function body and prototype changes

for C code analysis [3] that provides ASTs as well as some other high-level information about the source code. We have used it to analyze the complete lifetime of Vsftpd (a "very secure" FTP server, see `http://beasts.vsftpd.org/`) and OpenSSH (daemon); 8 snapshots in the lifetime of Apache 1.x; and portions of the lifetimes[1] of the Linux kernel (versions 2.4.17, Dec. 2001 to 2.4.21, Jun. 2003) and BIND (versions 9.2.1, May 2002 to 9.2.3, Oct. 2003).

Figure 6 shows the running time of the tool on these applications (we consider the tool's results below), plotting source code size versus running time.[2] The top line is the total running time while the bottom line is the portion of the running time that is due to parsing, provided by CIL (thus the difference between them is our analysis time). Our algorithm scales roughly linearly with program size, with most of the running time spent in parsing. Computing changes for two versions of the largest test program takes slightly over one minute. The total time for running the analysis on the full repository (i.e., all the versions) for Vsftpd was 21 seconds (14 versions), for OpenSSH was 168 seconds (25 versions), and for Apache was 42 seconds (8 versions).

3. CASE STUDY: DYNAMIC SOFTWARE UPDATING

This section explains how we used the tool to characterize software change to guide our design of a dynamic software updating (DSU) methodology [4]. We pose three questions concerning code evolution; while these are relevant for DSU, we believe they are of general interest as well. We answer these questions by using the output of our tool on the programs mentioned above, which are relevant to DSU because they are long-running.

Are function and variable deletions frequent, relative to the size of the program? When a programmer deletes a function or variable, we would expect a DSU implementation to delete that function from the running program when it is dynamically updated. However, implementing on-line deletion is difficult, because it is not safe to delete functions that are currently in use (or will be in the future). Therefore, if definitions are rarely deleted over a long period, the benefit of cleaning up dead code may not be worth the cost of implementing a safe mechanism to do so. Figure 7 illustrates how OpenSSH, Vsftpd, and Apache have evolved over their lifetime. The x-axis plots time, and the y-axis plots the number of function and global variable definitions for various versions of these programs. Each graph shows the total number of functions and global variables for each release, the cumulative number of functions/variables added, and the cumulative number of functions/variables deleted (deletions are expressed as a negative number, so that the sum of deletions, additions, and the original program size will equal its current size). The rightmost points show the current size of each program, and the total number of additions and deletions to variables and functions over the program's lifetime.

According to the tool, Vsftpd and Apache delete almost no functions, but OpenSSH deletes them steadily. For the purposes of our DSU question, Vsftpd and Apache could therefore reasonably avoid removing dead code, while doing so for OpenSSH would have a more significant impact (assuming functions are similar in size).

Are changes to function prototypes frequent? Many DSU methodologies cannot update a function whose type has

[1] Analyzing earlier versions would have required older versions of `gcc`.

[2] Times are the average of 5 runs. The system used for experiments was a dual Xeon@2GHz with 1GB of RAM running Fedora Core 3.

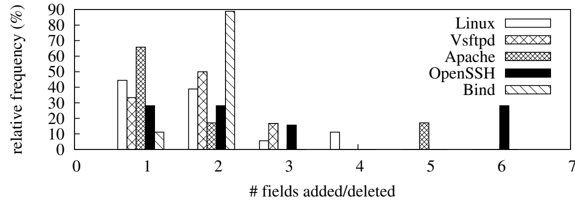

Figure 9: Classifying changes to types

changed. If types of functions change relatively infrequently, then this implementation strategy may be able to support a large number of updates. Figure 8 presents graphs similar to those in Figure 7. For each program, we graph the total number of functions, the cumulative number of functions whose body has changed, and the cumulative number of functions whose prototype has changed. As we can see from the figure, changes in prototypes are relatively infrequent for Apache and Vsftpd, especially compared to changes more generally. In contrast, functions and their prototypes have changed in OpenSSH far more rapidly, with the total number of changes over five years roughly four times the current number of functions, with a fair number of these resulting in changes in prototypes. In all cases we can see *some* changes to prototypes, meaning that supporting prototype changes in DSU is a good idea.

Are changes to type definitions relatively simple? In most DSU systems, changes to type definitions (which include `struct`, `union`, `enum`, and `typedef` declarations in C programs) require an accompanying *type transformer function* to be supplied with the dynamic update. Each existing value of a changed type is converted to the new representation using this transformer function. Of course, this approach presumes that such a transformer function can be easily written. If changes to type definitions are fairly complex, it may be difficult to write a transformer function.

Figure 9 plots the relative frequency of changes to `struct`, `union`, and `enum` definitions (the y-axis) against the number of fields (or enumeration elements for `enum`s) that were added or deleted in a given change (the x-axis). The y-axis is presented as a percentage of the total number of type changes across the lifetime of the program. We can see that most type changes affect predominantly one or two fields. An exception is OpenSSH, where changing more than two fields is common; it could be that writing type transformers for OpenSSH will be more difficult. We also used the tool to learn that fields do not change type frequently (not shown in the figure).

4. RELATED WORK

A number of systems for identifying differences between programs have been developed. We discuss a few such systems briefly.

Yang [5] developed a system for identifying "relevant" syntactic changes between two versions of a program, filtering out irrelevant ones that would be produced by `diff`. Yang's solution matches parse trees (similar to our system) and can even match structurally different trees using heuristics. In contrast, our system stops at the very first node mismatch in order not to introduce spurious name or type bijections. Yang's tool cannot deal with variable renaming or type changes, and in general focuses more on finding a *maximum syntactic similarity* between two parse trees. We take the semantics of AST nodes into account, distinguish between different program constructs (e.g., types, variables and functions) and specific changes associated with them.

Horwitz [1] proposed a system for finding *semantic*, rather than syntactic, changes in programs. Two programs are semantically identical if the sequence of observable values they produce is the same, even if they are textually different. For example, with this approach semantics-preserving transformations such as code motion or instruction reordering would not be flagged as a change, while they would in our approach. Horwitz's algorithm runs on a limited subset of C that does not include functions, pointers, or arrays.

Jackson and Ladd [2] propose a differencing tool that analyzes two versions of a procedure to identify changes in dependencies between formals, locals, and globals. Their approach is insensitive to local variable names, like our approach, but their system performs no global analysis, does not consider type changes, and sacrifices soundness for the sake of suppressing spurious differences.

5. CONCLUSION

We have presented an approach to finding semantic differences between program versions based on partial abstract syntax tree matching. Our algorithm uses AST matching to determine how types and variable names in different versions of a program correspond. We have constructed a tool based on our approach and used it to analyze several popular open source projects. We have found that our tool is efficient and provides some insights into software evolution.

We have begun to extend the tool beyond matching ASTs, to measure evolution metrics such as common coupling or cohesion [6]. We are interested in analyzing more programs, with the hope that the tool can be usefully applied to shed light on a variety of software evolution questions.

6. REFERENCES

[1] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 1990.

[2] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, Sept. 1994.

[3] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[4] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of the ACM SIGPLAN/SIGACT Conference on Principles of Programming Languages (POPL)*, pages 183–194, January 2005.

[5] W. Yang. Identifying Syntactic differences Between Two Programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[6] E. Yourdon and L. L. Constantine. *Structured Design, 2nd Ed.* Yourdon Press, New York, 1979.