Textual and Behavioral Views of Function Changes

Kai Pan, E. James Whitehead, Jr., Guozheng Ge
Dept. of Computer Science
Baskin Engineering
University of California, Santa Cruz
{pankai, ejw, guozheng}@cs.ucsc.edu

ABSTRACT
In this paper, we describe an approach that automatically computes function change information between consecutive revisions along the revision history of C language projects. Function changes are computed at two abstract levels. First, we compute the textual changes between two function revisions. Computed results include function additions and deletions, and the quantity and the ratio of textual change in changed functions across two revisions. Second, we compute the behavioral changes of functions using program slicing techniques. We use an XML-formatted document to represent computed function change information. The function change information, together with the SCM change log, helps maintainers understand code changes between two revisions. The structured format of the function change information also helps create traceability links between the changes and other artifacts. We describe our prototype implementation for computing function changes, and we evaluate our approach through a case study on the Sed project.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: Version control.

General Terms
Algorithms, Documentation.

Keywords
Version control, program slicing, program slice encoding.

1. INTRODUCTION
Software Configuration Management (SCM) tools such as CVS [1], Subversion [2], and ClearCase [3] are intensively used to perform version control and manage source code changes in modern software development projects. When submitting source code changes using these tools, best practice is for programmers to write a brief explanation of the change into the change log, which is saved together with the source code deltas in the SCM repository. After the source code of a software project has undergone many revisions, the change logs help maintainers to understand how the source code evolves, providing details on who changed which part of the code, and for what purpose.

Unfortunately, the quality of change logs varies greatly. Their quality depends on the programmers that submit the changes: how disciplined they are in making log entries, how well they understand the source code, and how well they write the change logs. Unclear or empty change logs make it difficult for software maintainers to understand the code. Even when they are present, the change logs are written in natural language and are mostly format-free. These facts make it hard to build traceability links based on information found in change logs.

To address the understandability and traceability issues in change logs, we propose an approach that automatically computes function-level change information across software revisions. Since functions are the basic functional block in C programs, we compute source code change information at the function level of granularity. Function change information has two abstract levels. The first level is the textual changes between two revisions of a function, which records the addition or deletion of text content. In our implementation, we use a C program analyzer to compute the text boundary of each function in the source code file of every revision. We retrieve the text body of a function and compare it with the text body of the same function in the previous revision using diff (GNU Diffutils) [4]. The diff results are used to check whether this function has textual changes in the new revision and compute the change ratio if new changes exist.

The second abstract level is behavioral changes of functions. We identify fine-grained function behavior aspects by using program slice information to compute the behavioral change of functions between two revisions. Program slicing techniques [5, 6] study the behavior of source code through the flow dependency and control dependency relationships among statements. The concept of program slicing was introduced by Mark Weiser [5], who defined a slice as all the program code that may influence the values computed at a criterion, which consists of a (line-number, variable) pair [5]. Horwitz et al. [6] introduced a method to compute intra-procedural and inter-procedural slices based on program dependence graphs. Since program slices capture the behavior aspects of a program, we can use them to identify the behavioral changes of functions across different revisions. A behavior aspect of a function can be represented by a program slice with respect to the *sensitive components* of a function at the end of the function. A *sensitive component* of a function is a variable that is or may be modified in this function and may be used by the other parts of the program. A function may have multiple *sensitive components*, such as the function return value, and the non-local variables directly or transitively modified in the function. We developed an algorithm, PSE (program slice encoding), that encodes a program slice to a hash value, so the change of a slice hash indicates the change of the behavior of this

program slice. We store the slice hashes for functions in each revision in the function change information to identify function behavioral changes across revisions.

Combining the information for function textual changes and the information for function behavioral changes, the function-level change information serves as complement to the change logs to help maintainers to understand the changes better.

We represent the function change information in XML format, so the change information can be identified at various granularities, including file, function, and function behavior aspect. The XML format also permits traceability links to be built between the change information and other software development artifacts.

2. FUNCTION TEXTUAL CHANGE

SCM tools such as CVS, Subversion, and ClearCase provide the capability to manage the text differences between two successive revisions of a file. *diff* [4] is a tool usually used to compute the text difference of two files based on line string matching. It calculates the longest common subsequence (LCS) of two text files at the line level.

The text differences computed by SCM systems are usually at the file level, even for source code files, since the *diff* tools used by SCM systems are general tools that are applicable to a broad range of programming languages and other artifacts, and hence they are unaware of source code syntax. However, to enhance the ability to understand source code changes, it is useful to compute change information at finer granularity than the file level, leveraging the software's syntactic structure.

We propose an approach that computes function textual change information between two revisions of C programs. This approach takes three steps.
Step 1: use a C program parser to compute the line range of every function in the source code files.
Step 2: find deleted and added functions by comparing the function lists of both revisions.
Step 3: for the functions existing in both revisions, we retrieve the function text from each revision and then compare them using GNU *diff*. This information is used to compute change information, including the number of lines changed and the ratio of line changes to the maximum number of lines of the function in the two revisions.

We compute the number of line changes and the ratio of line changes of two revisions of a function based on the output from GNU *diff*. GNU *diff* takes two arguments as input, which we term revision1 and revision2 according to input order. The output consists of a sequence with the following format:

change-command
< *from-file-line*
< *from-file-line...*

> *to-file-line*
> *to-file-line...*

The *change-command* consists of the line range of revision1, the change type, and the line range of revision2. The change type can be *a* for add, *c* for replace, or *d* for delete. For example, *5a3,5* means adding the lines 3-5 in revision2 after line 5 of revision1. *6,7d12* means deleting the lines 6-7 in revision1, and line 6-7 would have appear after line 12 in revision2 had they not been deleted. *10-12c18-21* means replacing line 10-12 of revision1 with line 18-21 of revision2.

We compute the number of line changes between two revisions of a function by scanning the *diff* results, and for every change type *a*, accumulating the number of added lines to the number of line changes; for every change type *d*, accumulating the number of deleted lines to the number of line changes; for every change type *c*, accumulating the maximum lines between the two line ranges in the change-command to the number of line changes. We compute the ratio of line changes by dividing the number of line changes by the maximum number of lines of the two revisions of the function.

3. FUNCTION BEHAVIORAL CHANGE

Though the function change information computed in Section 2 shows us the straightforward facts of the textual changes of a function in the new revision, no behavioral change facts are revealed. We know that some textual changes, such as changes to comments, code beautification, and variable renaming, do not affect program behaviors at all, while others may affect some aspects of program behaviors.

A behavioral aspect of a function can be represented by a program slice that computes the return value of the function, or a program slice that may modify a global variable in the function, etc. A function can have multiple behavioral aspects.

A program slice can be represented by a program dependence graph (PDG) [6]. To detect behavioral changes of two program slices, we compare their corresponding PDGs. To make the comparison process efficient and make the comparison results reusable, we developed a program slice encoding (PSE) algorithm that encodes a program slice to a hash value, thereby allowing the slice hashes of a function to be used to detect function behavior changes across revisions.

3.1 Program Dependence Graph

Our program slice encoding (PSE) algorithm works on program slices represented by the program dependence graph (PDG) defined by Horwitz [6]. A PDG is a directed graph, where each vertex in the PDG represents a statement or predicate in the program, and each edge represents a data dependence or control dependence relationship between the two statements corresponding to the two vertexes connected by the edge. Figure 1 shows an example program and its corresponding PDG.

As defined by Horwitz [6], there are three kinds of dependence edges in a PDG, *flow dependence*, *control dependence*, and *def-order dependence*. The *flow dependence* edges can be further broken down into *loop-independent flow dependence* and *loop-carried flow dependence*. The detailed definitions of these dependence edges can be found in [6].

3.2 The PSE Algorithm

Inspired by Horwitz's isomorphic-testing algorithm in [7], which checks the isomorphism of two PDGs, we developed the PSE algorithm to encode the PDG for a program slice. The basic process of the PSE algorithm starts with a depth-first graph traversal in the PDG of a slice, normalizing every node visited. Next, it encodes the transformed PDG to a string value, which is then fed to a hash algorithm to produce the final result, the hashed slice encoding.

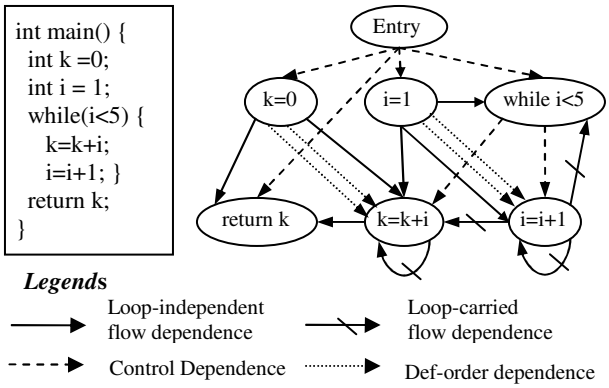

Figure 1. An example program and the program dependence graph for the slice with respect to *return k*.

Due to space limitations, we only explain the PSE algorithm for a simplified language that supports just assignment, if and while statements, and scalar variables. In our prototype implementation, we developed a full version of the PSE algorithm that works on C functions with arbitrary control flows, non-scalar variables, function calls and pointer variables. The basic idea is the same, though the full version of PSE copes with all advanced features present in C. There are four steps in the PSE algorithm.

Step 1 of the PSE Algorithm – Preprocess

This step follows the Preprocess step in the isomorphic-testing algorithm described in [7]. The purpose of this step is to mark every edge to make them distinct from one another in the incoming edges to a vertex. This permits a unique graph traversal path to be determined for the isomorphic PDGs to ensure the isomorphic PDGs are mapped to the same hash value.

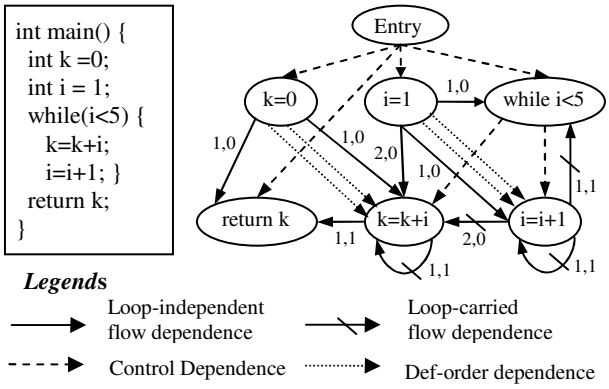

Figure 2. The program dependence graph with *operand numbers* and *ordering numbers* on flow dependence edges.

In this step, the *operand number* labels and the *ordering number* labels are added to the flow dependence edges in the PDG. An *operand number i* is added to a flow dependence edge from *u* to *v* if a variable *x* is defined by *u* and *x* is the i_{th} variable occurring from left to right in the expression at the right side of the assignment statement of *v*. The *ordering number* is defined to sort the incoming flow dependence edges that have the same *operand number*. In the example of Figure 2, the edge from *k=0* to *return k* has the same *operand number,* 1, as the edge from *k=k+i* to *return k*. But, since the statement *k=0* appears before the statement *k=k+i* in the program, we assign an *ordering number* of 0 to the edge from *k=0* and assign an *ordering number* of 1 to the edge from *k=k+i*.

Figure 2 shows the preprocessed PDG based on the original PDG in Figure 1. The first number on the flow dependence labels indicates the *operand number*, and the second number indicates the *ordering number*.

Step 2 of the PSE Algorithm – Normalize the graph

The second step of the PSE algorithm is to make a PDG walk in the preprocessed PDG in a certain order and normalize the variable names in the vertices. The PDG walk part in PSE follows the strategy used in the isomorphic-testing algorithm [7] to select the walk path in the PDG. The walk in the preprocessed PDG is a depth-first graph search starting from the slicing criterion vertex and then proceeding backwards along the flow and control dependence edges.

The order of the vertices visited in this depth-first PDG walk depends on the path selection rule that determines the traversal order of the incoming edges to a vertex. The detailed path selection rule is as follows:

(1) Take the control dependence edge first.

(2) Then, take the flow dependence edges by the order of their *operand number* from the smallest to the largest.

(3) For the flow dependence edges that have the same *operand number*, sort them by their *ordering number* from the smallest to the largest.

(4) For the flow dependence edges that have both the same *operand number* and the same *ordering number*, take the loop-independent flow dependence edge first, then the loop-carried flow dependence edges.

(5) For the loop-carried flow dependence edges that have the same *operand number* and *ordering number*, sort them by the loop nesting level of the loop predicate they carry from the most-deeply nested to the least-deeply nested.

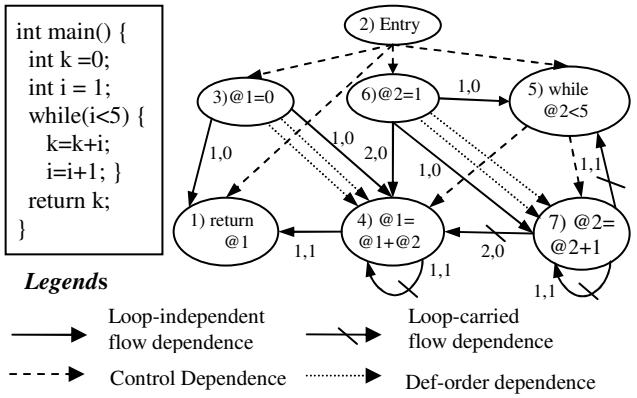

Figure 3. The program dependence graph after variable normalization.

During the depth-first walk in the PDG, the PSE algorithm processes every vertex by normalizing the variables in the statement. The PSE algorithm processes the vertex in this way:

(1) Make a preorder tree walk on the abstract syntax tree (AST) for the statement corresponding to the vertex.

(2) For every variable occurring in the AST walk whose name has not been processed, changed its name to a normalized name, @<*i*>, where <*i*> denotes that this is the i^{th} variable of all the variables renamed during the entire graph walk.

Figure 3 shows the PDG after a depth-first walk and variable normalization on the example PDG shown in Figure 2. The number added in every vertex indicates the order it was visited.

Step 3 of the PSE Algorithm – Encode the PDG

This step encodes the normalized PDG from Step 2 into a string value. This string value consists of two parts, the *statement encoding string* (encoding of statements) and the *edge encoding string* (encoding of dependence edges). The encoding process is as follows.

(1) Make a depth-first graph walk in this PDG in the same way as Step 2 of the PSE algorithm.

(2) For every vertex visited during the depth-first PDG walk, append the preorder format of the AST for the statement corresponding to the vertex to the *statement encoding string*.

(3) For every edge traversed, encode this edge to a string and append this string to the edge encoding string. The format of the encoded string for an edge is [a-b-c-d-e-f-g], where the fields *a* to *g* are values that represent the order number on the vertex at the source endpoint of this edge, the order number on the target vertex, the edge type, the operand number, the ordering number, the label value for the control dependence edge, and the loop nesting level of the loop predicate a loop-carried flow dependence edge carries.

After the encoding process in this step, the concatenation of the *statement encoding string* and the *edge encoding string* is the resulting encoding string for this PDG. For the example program in Figure 1, the *statement encoding string* is:

return (@1);{entry};=(@1 0);=(@1 +(@1 @2));while(<(@2 5));=(@2 1);=(@2 +(@2 1));

The *edge encoding string* is *[2-1-1-0-0-T-0][3-1-2-1-0-0-0][2-3-1-0-0-T-0][4-1-2-1-1-0-0][5-4-1-0-0-T-0][2-5-1-0-0-T-0][6-5-2-1-0-0-0][2-6-1-0-0-T-0][7-5-3-1-1-0-1][5-7-1-0-0-T-0][6-7-2-1-0-0-0][7-7-3-1-1-0-1][3-4-2-1-0-0-0][4-4-3-1-1-0-1][6-4-2-2-0-0][7-4-3-2-0-0-1].*

Step 4 of the PSE Algorithm – Hash the string value

This step maps the encoding string from Step 3 to a hash value using the MD5 hash algorithm [8]. The MD5 hash for the resulting encoding string from Step 3 is *6931bc735157676d42d1dc761c0fb357*.

The PSE algorithm has a linear runtime complexity of $O(m+n)$, where m represents the number of vertices in the PDG and n represents the number of dependence edges in the PDG.

4. IMPLEMENTATION

4.1 Project Architecture

To validate our ideas, we implemented a tool that computes the function textual changes and function behavioral changes of C programs across revisions. Our implementation uses the function line range information and procedure dependence graphs for function slices computed by CodeSurfer [9], a program analysis tool produced by GrammaTech, Inc.

To compute function textual change information across the entire revision history of a project, we check out every pair of consecutive revisions of the project from its Subversion repository, use CodeSurfer to identify all the functions in each revision, and compute their line ranges in the source code files. Based on the line range of the functions, we retrieve from each revision the function text and compare them using the *diff* tool. We also compute added functions and deleted functions in the later revision.

At the same time, we compute the function slice hashes for the functions in each revision. We use CodeSurfer to obtain the procedure dependence graph for intra-procedural program slices with respect to the *sensitive components* in each function, and apply the PSE algorithm on them. We compare the slice hashes of each function in the later revision with the corresponding hashes in the prior revision to find the function behavioral changes.

Finally, we save the function textual change information and function behavioral change information for each revision in a MySQL database. Figure 4 shows the project architecture and the flow of data among components.

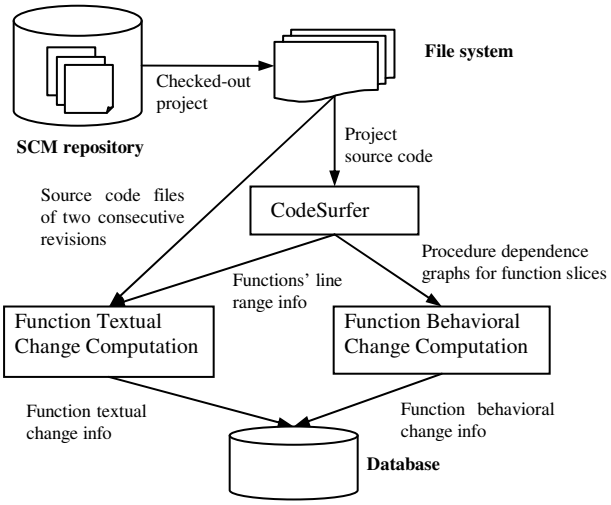

Figure 4. The project architecture and data flow.

4.2 Data Schema and Component Identification for Traceability

We represent the function textual change information and behavioral change information in XML format and store them in a MySQL database. The data in the database together with the change logs in the SCM repository will help maintainers understand the changes between revisions.

There is a *ProjectFunctionChange* table in the database. The table has three fields, which are *projectName*, *revisionNumber*, and *functionChangeInfo*. Each revision of the project has one record in the database. The *projectName* and *revisionNumber* fields record the name of the project and the revision number of the project respectively. The *functionChangeInfo* field contains the function textual change information and the function behavioral change information of this revision compared to its prior revision, represented in XML.

The XML DTD for the *functionChangeInfo* data is described as follows:

```
<!ELEMENT changeInfo (sourceFileChange*)>
<!ATTLIST changeInfo
        projectName CDATA #REQUIRED
        revisionNumber CDATA #REQUIRED>
<!ELEMENT sourceFileChange (functionChange*)>
<!ATTLIST sourceFileChange
        sourceFilePath CDATA #REQUIRED
        changeKind (Added | Deleted | Changed) #REQUIRED>
```

```
<!ELEMENT functionChange (sliceChange*)>
<!ATTLIST functionChange
        functionName CDATA #REQUIRED
        changeKind (Added | Deleted | Changed) #REQUIRED
        numberOfLineChanged CDATA #REQUIRED
        lineChangeRatio CDATA #IMPLIED>
<!ELEMENT sliceChange (sliceLabel, sliceHash)>
<!ATTLIST sliceChange
         sliceKind (Return | Parameter | Global) #REQUIRED
         changeKind (Added | Deleted | Changed) #REQUIRED>
<!ELEMENT sliceLabel (#PCDATA)>
<!ELEMENT sliceHash (#PCDATA)>
```

In the XML DTD, the element *changeInfo* represents all the changes to the project at this revision. The *changeInfo* element contains multiple *sourceFileChange* elements, which represent all of the deleted, added, and changed source code files. A *functionChange* element records the file name, change kind, number of line changes, the ratio of the line changes, and the change information for the function slices.

The *sliceChange* element records change in a behavioral aspect of the function. The slice can be one of the three types, *Return*, *Parameter*, and *Global*. A *Return* slice represents the behavior of computing the return value of the function. A *Parameter* slice represents the behavior of changing the value referenced by a pointer parameter of this function. A *Global* slice represents the behavior of changing the values of non-local variables. The *sliceLabel* element stores a label that indicates the detailed behavioral aspect the function slice represents. For example, if the function slice is for a global variable *foo*, its *sliceLabel* will be *foo*. Finally, the *sliceHash* element contains the 32-character hash value for the function slice computed by the PSE algorithm.

Due to the XML representation of the function change information, we open the door to locating components in the function change information and associating them with other artifacts such as test cases, bug reports, requirements, designs, requirement changes and design changes. We can use the following URI-style resource identifier syntax to locate components in the function change information:

functionChange://Project-name/version-number-or-range/
XPointer-in-functionChangeInfo

Below are two examples for the resource identifier expressions and their explanations.

Expression 1:

functionChange://Sed/6#xpointer(//functionChange[sliceChange[
sliceLabel="prog.cur"]])

Explanation: This expression selects all the functions in Sed revision 6 that have a changed behavior with respect to the global variable *prog.cur*.

Expression 2:

functionChange://Sed/versionRange(2,8)#xpointer(
//functionChange[@functionName="compile_address"
and @changeKind ="changed"])

Explanation: This expression locates all the changes to the *compile_address* function in the Sed project from revision 2 to revision 8.

The expressions that locate components in the function change information make it possible to create traceability links between the change information and other artifacts. Assume that we have the design document for the Sed project and it is represented in XML format, *sed-design.xml* (a UML document represented in XML). We locate the Sed design on the part that compiles an address for a Sed command using the expression:

sed-design.xml#xpointer(/compile/compile_sed_address)

After that, an XLink can be used to record the traceability between the design component for *compile_sed_address* and the function change information for the *compile_address* function, which is an implementation of the design component:

```
<cinfo_link xlink:type="extented">
    <loc xlink:type="locator"  xlink:href='sed-design.xml#
                    xpointer(/compile/compile_sed_address)"
            xlink:label="design_anchor_1"/>
    <loc xlink:type="locator"
            xlink:href=" functionChange://Sed/versionRange(all)#
            xpointer(//functionChange[@functionName=
                'compile_address'])"
            xlink:label="changeInfo_anchor_1/>
    <link xlink:type="arc" xlink:from=" design_anchor_1"
            xlink:to="changeInfo_anchor_1"/>
</cinfo_link>
```

To support the traceability, we still need to provide a implementation to parse and retrieve change information from the special URL, *functionChange://Project-name/version-number-or-range/*, in XLinks.

If the design document changes can be represented in XML too, XML links can also be created between the design changes and the function change information for source code, so we can trace from the source code changes to the design component changes to find out the reason of source code changes.

5. CASE STUDY

We tested our implementation on revision 3.01 and revision 3.02 of the Sed project [10]. Sed revision 3.01 has 12223 lines of code, 12 C files and 100 functions and Sed revision 3.02 has 12368 lines of code, 12 C files and 100 functions.

We made a function text comparison for Sed source code between revision 3.01 and revision 3.02. The results show that only the *snarf_char_class* function in *compile.c* has textual change: 2 lines in revision 3.01 are replaced by 5 lines in revision 3.02. The *snarf_char_class* function in revision 3.01 has 42 lines and it has 45 lines in revision 3.02. So, the number of line changes for *snarf_char_class* between 3.01 and 3.02 is 5, and the ratio of line changes is 11%.

We also made a function behavior comparison for Sed between revision 3.01 and revision 3.02, the results show that the function *snarf_char_class* in revision 3.01 has four slices: one *Return* kind, one *Parameter* kind, whose *sliceLabel* is #param1, one *Global* kind whose *sliceLabel* is *prog.cur*, and the other *Global* kind whose *sliceLabel* is *input.line.* Function *snarf_char_class* in revision 3.02 has four function slices with the same labels too, but all of their slice hashes are different from those in revision 3.01, which means the textual changes in the *snarf_char_class* function affect all the four behavior aspects of this function.

In summary, the *functionChangeInfo* data for Sed revision 3.02 is as follows.

```
<changeInfo projectName="sed" revisionNumber="3.02">
 <sourceFileChange sourceFilePath="sed/compile.c"
                    changeKind="changed">
  <functionChange functionName="snarf_char_class"
          changeKind="changed"  numberOfLineChanged = "5"
          lineChangeRatio="0.11">
```

```
    <sliceChange sliceKind="Return" changeKind="changed">
      <sliceLabel>#return</sliceLabel>
      <sliceHash>ce2d52d65f33eb611a6030735ebe9262</sliceHash>
    </sliceChange>
    <sliceChange sliceKind = "Parameter" changeKind="changed">
      <sliceLabel>#param1</sliceLabel>
      <sliceHash> 3a2289b2f656d5569ea0110d07f8a1c5</sliceHash>
    </sliceChange>
    <sliceChange sliceKind = "Global" changeKind="changed">
      <sliceLabel>prog.cur</sliceLabel>
      <sliceHash>f6efe3e368291a4f5dcb6b21502219c8</sliceHash>
    </sliceChange>
    <sliceChange sliceKind = "Global" changeKind="changed">
      <sliceLabel>input.line</sliceLabel>
      <sliceHash>f6efe3e368291a4f5dcb6b21502219c8</sliceHash>
    </sliceChange>
  </functionChange>
 </sourceFileChange>
</changeInfo>
```

6. RELATED WORK

Computing document difference based on document syntax or structure has been explored by many research efforts. Yang [11] developed a syntactic comparison and merge tool based on parse trees for the C programming language. Different files are parsed to generate corresponding parse trees. Then, a tree matching algorithm runs to match nodes and locate differences. Finally, a pretty-printer traverses the trees and highlights the different code sections in the files. Cdiff [12] takes a similar approach for C++. Due to the limitation of the C++ Information Abstractor cdiff uses, it only handles comparison at the procedure level. There are also syntactic comparison approaches that use graph structure to represent the code. For example, [13] uses labeled typed nested graphs and graph rewriting techniques to provide a formal foundation for software diff and merge. Our approach that computes the function textual changes combines the use of syntax analyzer and text diff. Our approach relies much less on the understanding of the full program syntax, since only the syntax units at the function level are retrieved, the text diff is still heavily used in our approach.

The program behavior comparison problem has been explored somewhat. Horwitz and Reps [7] introduced an algorithm to compare two program slices. Their algorithm is based on the dependence graph representation of program slices. The labels on the dependence edges in the graph and the dependence type between graph vertices guide the comparison of two slices. Though our PSE algorithm is based on Horwitz's algorithm in [7], our algorithm requires only one entity, a program slice, and maps it to a hash value, which is reusable for comparing behaviors across revisions. In [14], Apiwattanapong et al. introduced the CalcDiff algorithm that compares the behaviors of object-oriented programs. To compare program behaviors, their approach employs enhanced control flow graphs, which neither address variable renaming and statement permutation, nor extract finer-grained behavior aspects from a method.

7. CONCLUSION AND FUTURE WORK

In this paper, we describe an approach that computes function change information for C projects across revisions. Two levels of abstraction, function textual change and function behavioral change, are included in the function change information. We compute function textual changes between two revisions by retrieving function text from these revisions and comparing their difference using the *diff* tool. We capture function behavior aspects using program slicing techniques. We introduce the PSE algorithm that encodes a program slice to a hash value. We apply the PSE algorithm on C functions by computing the hash values for program slices with respect to *sensitive components* of a function. The slice hashes for a function can be used to identify function behavioral changes across revisions. The function textual change information and the function behavioral change information serve as complement to the change log to help maintainers understand program changes across revisions.

We represent the function change information with the XML format and define the syntax to locate components in the XML. This representation of change information opens the door for creating traceability links between the function change information and other software artifacts.

For the future work, we will integrate our approach with an SCM system, Subversion, so function-level change information can be stored in SCM change logs and, at the same time, it can guide the change submitter to write change logs in structured way.

8. REFERENCES

[1] S. Dreilinger, "CVS Version Control for Web Site Projects," 1998, http://interactive.com/cvswebsites/.

[2] Subversion, "Subversion Project Home Page," 2005, http://subversion.tigris.org/.

[3] IBM, "Rational ClearCase Home Page," 2004, http://www-306.ibm.com/software/awdtools/clearcase/.

[4] GNU, "GNU Diffutils," 2003, http://www.gnu.org/software/diffutils/.

[5] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352-357, 1984.

[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[7] S. Horwitz and T. Reps, "Efficient Comparison of Program Slices," *Acta Informatica*, vol. 28, no. 9, pp. 713 - 732, 1991.

[8] R. Rivest, "The MD5 Message-Digest Algorithm," 1992, http://www.ietf.org/rfc/rfc1321.txt.

[9] GrammaTech, "GrammaTech CodeSurfer Home Page," 2004, http://www.grammatech.com/products/codesurfer/index.html.

[10] GNU, "GNU Sed (streams editor)," 2003, http://www.gnu.org/software/sed/.

[11] W. Yang, "How to Merge Program Texts," *Journal of Systems and Software*, vol. 27, no. 2, 1994.

[12] J. E. Grass, "Cdiff: A Syntax Directed Diff for C++ Programs," In Proceedings of USENIX C++ Conference, Portland, OR, 1992, pp. 181-193.

[13] T. Mens, "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution," in *LNCS*, vol. 1779: Springer-Verlag, 1999, pp. 127-143.

[14] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," In Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04), 2004, pp. 2-13.