Change Impact Graphs: Determining the Impact of Prior Code Changes

Daniel M German  
University of Victoria  
dmg@uvic.ca

Gregorio Robles  
Universidad Rey Juan Carlos  
gregorio.robles@urjc.es

Ahmed E. Hassan  
Queen’s University  
ahmed@cs.queensu.ca

Abstract

The source code of a software system is in constant change. The impact of these changes spreads out across the software system and may lead to the sudden manifestation of failures in unchanged parts. To help developers fix such failures, we propose a method that, in a pre-processing stage, analyzes prior code changes to determine what functions have been modified. Next, given a particular period of time in the past, the functions changed during this period are propagated throughout the rest of the system using the dependence graph of the system. This information is visualized using Change Impact Graphs (CIGs). Through a case study based on the Apache Web Server, we demonstrate the benefit of using CIGs to investigate several real defects.

1. Introduction

All too often when maintaining a large software system, a bug report is submitted regarding changes in the behavior of an unchanged functionality. Investigating this type of bug reports is difficult and tedious, since the fix is frequently in a different location than the location where the failure manifests itself, i.e., the reported location in the bug report. The failing behavior is usually due to the ripple effect of another change in a different part of the system that propagates along various dependencies, such as call and data dependencies, and affects the unchanged code.

The maintainer in charge of fixing such failures starts her investigation with the location where the failure manifests itself. She then examines the dependency graph of the reported failing function in an ad-hoc manner using her knowledge and her experience about the software system trying to pin down the actual location of the bug causing the failure. A maintainer could use slicing techniques [22, 21] to determine all the code locations which may affect the reported location of a failure and are likely the source of the bug causing the failure. However, slicing techniques are known to report large slices [4, 3]. A single slice may contain as much as as 30% of the source code of an application. Maintainers would spend considerable time investigating such large slices for complex real-life software systems. Approaches, such as dynamic slicing [1, 24], have been proposed in literature to reduce the size of slices and make them more accurate for large software systems. However most techniques require additional effort (e.g., execution of tests for dynamic slicing) and expensive analyses.

In this paper, we propose a method which determines the impact of historical code changes on a particular code segment (e.g., a function). Given the reported location of a failure, a maintainer is particularly interested in being aware of recent code changes which could have impacted the functionality of the failing function—specially if that function was not changed recently. Our method determines all the part of the software system which affect the reported location of a failure. The method then annotates these parts by marking recent code changes and propagating the impact of these recent changes. It then creates a change impact graph to determine what areas might have been affected by certain changes to help maintainers rapidly pinpoint the source of a bug given the reported location of a failure. The maintainer needs to only examine the marked up functions instead of going through all the functions which would be produced by a slicing technique.

We demonstrate the feasibility and possibilities of our method with a case study using the call-graph information. Our case study uses several real bug reports from the Apache Web Server and demonstrates the benefit of using our method to investigate the reported failures and fix the corresponding bugs.

Organization of the Paper

The remainder of the paper is organized as follows: the next section introduces the model used to track the impact of historical code changes. Section 3 presents the methodology used to analyze historical code changes and recover their impact on source code entities (i.e., functions). Section 4 presents a case study of using our method to fix three real bugs from the Apache Web Server using our proposed method. Section 6 discusses the effectiveness, limitations,
and possible improvements for our method. Section 7 concludes the paper.

2. A model to track the impact of historical code changes

Historical changes to a function can be modeled as a sequence where each element corresponds to the source code of the function after each particular change. Formally, for a function \( f \) we define its change history sequence as \( H_f = (f_0, ..., f_m) \), where \( f_i \) is the \( i \)-th instance of the function. Each element, i.e., instance of a function, can be annotated with metadata about the change such as the date of the change, the name of the developer who performed the change, and the purpose of the change.

The change history sequence of \( f \) can be represented as a graph. She is only aware of the edges that start in \( f \) and its edges are the direct calls between any of these functions. If a function \( g \) is called from function \( f \), the dependence graph of \( f \) contains the dependence graph of \( g \). The dependence graph can be considered a simplified file dependency graph that tracks function invocations and does not track parameters nor variables [21]. The dependence graph of \( f \) includes any function that could be called by \( f \), including constructors, destructors, and library functions. When a developer peruses the source code of a function \( f \), she is not usually aware of all the contents (or the size) of this dependence graph. She is only aware of the edges that start in \( f \) (the function calls inside \( f \)).

The dependence graph of a function \( f \) can be created at any time \( t \) during the life of such a function. The graph is built recursively as described above, using the latest instance of every one of all the functions in the graph such that their date of modification is less or equal to \( t \). In other words, if we want to build the dependency graph of \( f \) on Dec. 31, 2007, then we will use the latest instance of \( f \) with a date less or equal to Dec. 31, 2007. If it calls a function \( g \) then we will use the latest instance of \( g \) with a date less or equal to Dec. 31, 2007. This process will continue until the dependence graph is completed.

The dependence graph of a software system is the union of the dependence graphs of all its functions.

We illustrate our model with a simple example. Assume a C source file that has had four changes recorded as depicted in Figure 1. The change history for its functions is shown in Figure 2. The change history tracks when the functions are added, deleted or modified.

![Figure 1](source_code.png)

**Figure 1.** Evolution of the source code of an example system at four different points in time. The areas affected by each change are shown in bold.

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>A</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>c</td>
<td>A</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>A</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2](change_history.png)

**Figure 2.** Depiction of the change history for the functions of the example system. The rows correspond to the functions and the columns to the changes. \( A, M, D \) are respectively, Added, Modified and Deleted. \( exit \) is not included because it is an external function.

2.1. Propagation of prior changes

A typical use-case involves a developer who is perusing the source code of function \( f \) at time \( t \), and who is interested to know any changes that might have had an impact on the behavior of \( f \) during a particular time window \([t_b, t_e]\) (note that the period of interest does not need to include \( t \), e.g. the graph can be created in December with changes performed during April to May – the period of interest). To answer such a question, the dependence graph of \( f \) is computed at time \( t \) and its nodes are annotated according to any changes during the period of interest \([t_b, t_e]\) as follows:

1. Mark all nodes in \( G(f) \) as unaffected.
2. For each node \( g \) in \( G(f) \): if it has been added or changed during \([t_b, t_e]\), then annotate it as \textit{changed}.

3. Repeat until the graph no longer changes:
   - for any node that is still \textit{unaffected}, mark it as \textit{affected} if at least one of its children is either \textit{changed} or \textit{affected}.

In the resulting dependence graph (which we call a \textit{Change Impact Graph} or CIG) each node will be of one of three types:

1. \textit{Unaffected}. The function nor any of the functions it can potentially call were affected by the changes.
2. \textit{Changed}. The source code of the function has been changed.
3. \textit{Affected}. The source code of the function has not changed, but at least one of the functions it can potentially call has changed.

Figure 4 shows the CIG for the example of Figures 1 to 3. Notice how the change in \( d \) and \( e \) is propagated to \( b \) and \( a \) (the functionality of both of these functions might be affected), but not to \( c \).

### 2.2. Quantifying the impact of changes

We define two metrics to quantify the effect of the changes during a period of interest: the \textit{ratio of changed functions} and the \textit{ratio of affected functions} in the CIG of a function.

- The \textit{ratio of affected functions} is the proportion of \textit{changed} and \textit{affected} to total nodes in a CIG (of a function or a system). It provides an overview of the area impacted by the changes. If a set of changes have a large ratio of affected functions, then such changes have the potential to affect the functionality of a large proportion of the functions in the software system. Using our running example shown in Figure 4, the ratio of affected functions is \( 4/5 \).

- The \textit{ratio of changed functions} is the proportion of \textit{changed} nodes to total nodes in a dependence graph. This ratio gives an overview of the proportion of functions changed. Using our running example shown in Figure 4, the ratio of changed functions is \( 2/5 \).

In practice, the higher the ratio of affected functions is, the more areas a failure-inducing change could affect. By computing the ratio of affected functions of a potential change a developer could assess how critical a change is.

When a developer computes a CIG, she will want to minimize the ratio of affected and changed functions. She will usually work with the current version of the source code, and specify a period of interest in the past. She will want to narrow potential areas of the code that would have been affected during such changes. The longer the historical period, the higher the ratio of changed functions, making this method less effective. The major challenge of this proposed method of dependence graph annotation is to find a suitable period of interest such that the buggy change which introduced the failure (or any other interesting functionality) is within it, while minimizing the ratio of changed functions.

### 2.3. Annotating Source Code

Dependence graphs of real systems are usually complex and difficult to read or visualize. We propose instead to annotate the source code of any function with the help of the CIG. In its most simple conception, each line of code will
be tagged if it contains a call to a function that is marked affected or changed (we call this the impact-annotated source code). Figure 5 shows the source code of functions \( a \) and \( b \) after change \( C3 \); the calls for both functions have been colored according to their CIGs (for the same period of interest) as depicted in Figure 3. The color scheme is the same as the one used in the CIGaffected functions are shown in blue, and changed functions in red. The color of the calls give awareness to the developer that during the period of interest there were changes that affected \( b \), and both \( d \) and \( e \) were changed.

```c
void a() {
    b();
    c();
}

void b() {
    d();
    e();
}
```

Figure 5. Impact-annotated source code of functions \( a \) and \( b \) after change \( C3 \) for period for changes \( C1 \) and \( C2 \). Neither function changed during the period, but the functions they call (\( d \) and \( e \)) did change.

Let us assume that a failure was reported in \( a() \) after \( C2 \), and that this failure did not exist before \( C1 \). In other words, the failure is presumed to have been caused by a bug introduced during changes \( C1 \) or \( C2 \) (or both –the graph presented in Figure 5 was created after \( C2 \) and its period of interest includes \( C1 \) and \( C2 \) only). The developer will probably start by inspecting function \( a() \). The \( c() \) function is not likely to be the cause of the failure (it is not changed nor affected) and could be ignored (or at least presumed to have a lower probability of being the location of the bug). On the other hand, function \( b() \) is marked as affected, so it is worth exploring function \( b() \) to see if the change to one of the functions in its dependence graph has introduced the bug. The goal of highlighting affected and changed lines of code is to guide the attention of the developer towards the functions that are more likely to be responsible for a failure.

3. Recovering the impact of function evolution from a version control system

In this section we present the implementation of the model described in Section 2. We assume that the source code history is stored in a version control system (such as subversion or CVS). Although we discuss our implementation within the scope of C, it is could be extended to other programming languages with minimal effort.

3.1. Recovering the change histories of functions

We use the information recorded in the version control system to compute the history of each function. Since version control systems track the evolution of a software system at the line level instead of the function and dependency level, we need to perform additional analysis and extraction in order to recover the history of code changes and the function and dependency level.

For each version of every source code file in the history of the system, including those files that have been deleted, we recover the content of each instance of each function using the following technique:

- We identify the location where the definition of a function starts. We use the exuberant ctags\(^1\) tool for this purpose.
- For each function defined in a file, we look for its ending location. The end of a function is assumed to be the location of the last closing brace before the next definition. When processing C source code we do not consider macros as a definition, as macros can appear in the middle of a C function.

Using the content of each instance of a function, we can map each source control change to the particular function. However, whitespaces or comments of functions are sometimes changed. For example, PostgreSQL reformats its source code on a regular basis [9]. We do not want to consider these changes since they do not have any effect on the functionality. We developed a technique to remove comments using the mangle tool; then we re-indent the source code for each instance of a function using the indent tool.

We proceed to compare each instance of a function to its predecessor, resulting in a list of unchanged, modified, added, and deleted functions. We identify each function uniquely by the filename where it is found, and its name. This approach permits us to deal with multiply-defined functions such as local static functions.

One major challenge is the detection of functions that have been moved and/or refactored. It is interesting and valuable to have a precise picture of the history of a function, but this analysis is not required for our method. Our main goal is to provide awareness of changes, i.e., to know that the dependence graph of a function has changed, not necessarily how it has changed. Our method could be extended using one of several methods to recover renaming and refactoring, such as the ones described in [23, 14] (we discuss this issue further in section 6.4).

We annotate each instance of a function with the metadata of the change, such as the date and the developer. Finally, the change histories of each function that has been present in the system are stored in a relational database.

\( ^{1}\)ctags.sourceforge.net
3.2. Creating the Change Impact Graph (CIG)

In addition to recovering the change history of each function in a software system, we need to create the CIG of the software system at any moment in time. We used the following technique to create the dependence graph at a given time:

- Using the version control system, we retrieve the source code of the project as it was at a desired time.
- For each function in each source code file, we obtain its explicit calls to other functions. We use CCFinder [13] to extract the ASTs of each source file. From these ASTs we get the direct calls to other functions. The CIG of any function is created and annotated using the algorithms described in Section 2.

4. Case Study

We performed a case study to evaluate the feasibility of our method and to investigate its possibilities and limitations. For our study, we used the Apache Web Server version 1.3. We selected Apache for several reasons: it is a large, complex and well-known software system with a rich history and a large number of developers. In addition, its defect tracking database is publicly available.

Although version 1.3 is currently in its maintenance phase, it is still widely in use. It has approximately 86 kSLOCs and is mostly written in C. It has 8,021 commits and 29,999 revisions. A commit is a logical transaction that consists of one or more changes, i.e., revisions, to a file. More information about Apache, its community and its way of development can be found in [16]. We made a copy of its subversion repository to avoid overloading Apache’s servers.

To demonstrate the usefulness of our method, we needed to identify historical code changes that resulted in the manifestation of a failure in a different area of the software system. We searched the source control system for description of changes (the commit logs) which included the words “introduced”, “bug” and “PR” followed by a number. Changes which fix a bug in Apache usually include a reference to the bug in the defect system using the following syntax: PR #<number>. We located seven such changes. We selected the three most recent changes. These changes fixed the following bug reports: PRs #3130, #5389, #10090 and #10185. These reports are depicted in Table 1.

Our goal was to determine if we could use our method to identify the prior code change, which may have introduced the bug that caused the reported failure, given the location of a failure as documented in the bug report. For each failure location, we used the time of the reported failure in the

---

2We use the tokenizer of ccfinder to extract the calls. Hence we are capable of dealing with explicit calls only.
Directories have size shown as “0k” instead of “-” in Fancy Heading.

mod_rewrite is *SEVERELY* broken by a one-character bug introduced in version 1.148. The bug causes the next-to-last backref substitution to never happen... if you only have one backref, the $1 disappears without a trace!

PRs #10090 and #10185 These two bug reports documented a failure which also affected the rewrite module (mod_rewrite.c). The submitter of one of these reports claimed that a change between versions 1.3.22 (Oct 12, 2001) and 1.3.23 (released Jan 24, 2002) had broken the “rand map type”. The failure was reported March 14, 2002. As shown in Figure 10 only four functions of the dependency graph of the hook_uri2file function (the main entry point of the module) were changed during those two dates. One of the four functions: the rewrite_rand function is the location of the bug. This change took place on Jan 20, 2002, just 4 days before the release of 1.3.23.

The impact-annotated source code of rewrite_rand is presented in Figure 9. The error was introduced when a developer added the typecast (int) to the front of the expression; the priority of this operator applied the typecast to the denominator of the expression only. The log of this change reads: “Dispatch 26 compiler emits into oblivion. Vetting
is desired, please post to the list if you participate. They are all blindingly obvious, but extra eyes always help. This eliminates all but the regex emits and MSVC’s borked mis-declaration of FD_SET.”.

Changes like these are probably riskier than traditional changes because they are done in mass (26 compiler errors fixed in one change). It is clear that the developer did not fully test this change. Otherwise the bug would have been discovered almost immediately; instead it resulted in a failure almost three months after the bug was introduced.

Impact-annotations can be very useful in these situations, because people affected by any of these changes will know it and might be more inclined to check it for correctness. Otherwise, as in the case of this bug, nobody reviewed this line of code (or if it was reviewed, the reviewer failed to catch the bug).

5. Related Research

Change propagation is a central activity during software development. As developers modify code to introduce new features or fix bugs, they must ensure that other parts of the software system are updated to be consistent with these new changes. For example, if the interface for a function changes, its callers have to be modified to reflect the new interface, otherwise the source code won’t compile nor link.

Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly. Our proposed method provides a practical and simple technique which mine historical code changes to help maintainers in fixing bugs caused by mis-propagation of changes.

The dangers of mis-propagating changes has been noted by many researchers. For example, Parnas tackled the issue of software aging and warned of the ill-effects of Ignorant Surgery, code changes done by developers with limited knowledge of the system [18]. Arnold and Bohner give an overview of several formal models of change propagation [2, 5]. The models propose several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure [21, 22] to assist in code propagation. Rajlich proposes another formal model for change propagation [19]. In contrast, we propose a simplified practical model and implementation which developers can use to
static int rewrite_rand(int l, int h) {
    rewrite_rand_init();
    /* Get [0,1) and then scale to the appropriate range. Note that using
       * a floating point value ensures that we use all bits of the rand()
       * result. Doing an integer modulus would only use the lower-order bits
       * which may not be as uniformly random. */
    return (int)((double)(rand() % RAND_MAX) / RAND_MAX) * (h - l + 1) + l;
}

Figure 9. Annotated source code of rewrite_rand_init. Its first source code line was not modified
nor affected; the second—the cause of the failure—was modified on Jan 20, 2002, when the typecast
operator (int) was inserted. The log of the change explains: “Dispatch 26 compiler emits into
oblivion. Vetting is desired... They are all blindingly obvious, but extra eyes always help...”.

identify possible mis-propagation of changes when working
on fixing bugs.

Several researchers have proposed the use of historical
data related to a software system to assist maintainers of
large software systems. Cubranic et al. present a tool which
uses bug reports, news articles, and mailing list posting to
suggest pertinent software development artifacts [7]. Chen
et al. attach the comments associated with source code
changes to each code statement and use these comments to
index the code and help in locating the lines of code associ-
ated with a particular feature [6]. Hassan and Holt propose
annotating the dependency graph of a software system with
historical information to assist architecture in understanding
the rationale for the current design [11]. Mockus et al. use
historical code changes to help identify code experts based
on prior changes for a particular code segment [17]. Rela-
tive to previous work on the use of historical information
we recognize the importance of historical information and
we integrate the historical information into the commonly
used dependency information (i.e., the dependence graph).

Much of the intuition and driving force behind our work
stems by the following two related works. Graves et al.
show that surprisingly most bugs are not due to complex
code instead they are usually due to frequently changing
code [10]. Given the location of a reported bug, our method
flags statements which depend directly or indirectly on
changing code. Sliwerski et al. present a procedure which
identifies risky code regions using information from version
history and from the bug tracking system [20]. They present
an Eclipse plug-in which informs developers about the
risk of a location on a statement basis. The risk is calculated
based on the number of times a particular statement was
part of a change that was later identified as being a buggy
change. Similar to Sliwerski et al., our method helps de-
vlopers identify risky parts of the code. In contrast, our
definition of risk is a second-order definition: instead of
identifying risky code, we identify code that depends on
risky code by, for example, calling code which tends to have
many buggy changes.

6. Discussion

6.1. Limitations

Figures 6 and 7 computed for PR #3130 illustrate two of
the major shortcomings of our method: 1) a single change
can result in too many marked nodes in the dependence
graph that it becomes impractical; and 2) it is sometimes
not easy to determine the period of interest for which the
dependence graph should be created. Table 2 gives an ex-
ample of this shortcoming. The first graph had a ratio of
changed functions of 81.7%, and the second had a ratio of
8.8%. The developer needs to experiment and apply her
experience and insight in the selection of the period of in-
terest. In the case study, we removed a wide change that
had affected most of the functions in the system.

Systems with a very good suite of tests will benefit from
CIGs. Failures are likely to be found early, making the pe-
riod of observation very small. The annotations will point to
the few areas of the system that are likely to have changed
in such a small period.

Another method to deal with changes that affect many
functions is to select only a subset of changes based on cer-
tain criteria – as described in [15]. For example, “select
all commits during the period of observation except the one
that renamed all symbols”. The risk of using this method
is that one might inadvertently skip the commit that intro-
duced the bug which caused the reported failure. This is not
an issue when one is interested only in being aware of what
areas of the system have changed (and which have been af-
fected). For example, a developer might be interested to get
an idea of what areas have been affected by the changes per-
formed by another developer; in this case the criteria is to
select only the changes authored by the latter author.
6.2. Extraction of the Dependence Graphs

The effectiveness of CIGs depends heavily on the quality of the extraction of the dependence graphs created from the source code of the system. In our current implementation we use a simple fact extractor that does not take into account function pointers nor polymorphic function calls. Our method to create CIGs, however, can work with any dependence graph extractor that generates a graph where functions are represented as nodes, and function calls as edges.

6.3. Effectiveness of CIGs

The examples above are too few and lack the necessary rigor to be considered a formal evaluation of CIGs. Nonetheless the examples demonstrate that dependence graphs can narrow the search space for the source of a failure by highlighting areas of the code that have changed and that might have an impact on a failing function. Table 2 shows the ratios of changed and affected functions for each of the CIGs presented in our case study. Although the CIGs are relatively large, the ratio of changed nodes is very small (as small as 2.3%) for three out of the four graphs.

However even if a graph contains few changed nodes, the number of affected nodes (functions where a failure can occur) can be large. In other words, a bug introduced in a function has the potential to present itself as a failure in many other functions.

<table>
<thead>
<tr>
<th>PR</th>
<th>Total nodes</th>
<th>Ratio</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>#3130</td>
<td>142</td>
<td>88.7%</td>
<td>81.7%</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>17.7%</td>
<td>8.8%</td>
</tr>
<tr>
<td>#5389</td>
<td>206</td>
<td>46%</td>
<td>2.3%</td>
</tr>
<tr>
<td>#10090,#10185</td>
<td>75</td>
<td>30%</td>
<td>5.3%</td>
</tr>
</tbody>
</table>

Table 2. Effectiveness of CIGs for the examples presented in our case study.

6.4. Improving the tracking of a function’s evolution

Some functions are renamed, merged, split or their code cloned. We believe it will be worthwhile to track this evolution and use the resulting information in the creation of CIGs.

Similarly, the analysis we present relies on a textual comparison (comments are removed, code re-indented and then renamed). A more powerful approach would involve comparing the ASTs of the function and after the change (using methods such as [8]). Did the change affect the AST of the code? Was it a change to a constant (such as a string to be printed)? Was it a change to a token (perhaps the result of a rename of a function in the same commit). This information could be useful to include and exclude some changes when building a CIG.

6.5. Improving the annotations

The changed functions of the dependence graph can be further annotated with a measure of the change, such as the number of LOCs changed, the difference of the complexity between before and after, a likelihood that the change is a risky one (based on the type of change, who made the change, when the change was performed, etc.). Such information can then be propagated to the callers.

6.6. Support for annotations during editing/debugging of source code

The annotated source code could be computed on-demand within a typical IDE (such as Eclipse) or a debugger. In a preparation stage, the history of the project is analyzed, and the change history of each function is created. The latest dependence graph of the system is computed. At this point it is possible to incrementally continue updating the change histories of functions and the latest dependence graph as the version control system detects a new source control change. When perusing source code, the developer will select the period of interest (either by time, or by specifying two different changes). If the code being browsed has not changed (with respect to the latest version in the source control repository), the pre-computed CIGs would be used, otherwise a new one will be computed. The source code will be annotated using these CIGs.

6.7. Annotation of other program representations

It is possible to apply the same annotation method to more complex program representations, such as system dependence graphs that track variables and in-and-out parameters to functions [12]. Code slicing will provide a more in-depth analysis of impact of the changed code. Slicing will track indirect calls via parameters, and changes to variables, in contrast to our method which is based only on tracking function calls. Historical annotation will likely highlight small areas of a slice, making them more manageable when a developer is looking for a particular bug. The annotations of the slice could be applied dynamically by allowing the developer to change the period of observation as she finds it appropriate.

7. Conclusions

All too often developers must investigate failures in functions and features that have not changed. Investigating
such failures is challenging and time consuming since these failures are usually due to bugs introduced by prior code changes. In this paper we present a technique which guides developers in their investigation of such failures by annotating the dependence graph and the source code of a function with the impact of prior historical changes. Using the annotation, developers can quickly pinpoint the changes which most likely introduced the bug, causing the reported failure.

We demonstrate the feasibility of our method through a case study on the Apache Web Server. Our method speeds up the process of identifying the location of bugs by considerably reducing the number of functions which should be investigated.

8. Acknowledgements

We would like to thank our anonymous reviewers for their helpful comments.

The work of D. German and A. Hassan is funded in part by the Natural Sciences and Engineering Research Council of Canada. The work of G. Robles has been funded in part by the European Commission, under the FLOSSMETRICS project, and by the Spanish QUALIPSO (FP6-IST-034763) projects, and by the Spanish QUALOSS (FP6-IST-5-033547) and by the Spanish ICyT, project SobreSalto (TIN2007-66172).

References


