Automatically Documenting Program Changes

Raymond P.L. Buse and Westley Weimer*
The University of Virginia
{buse, weimer}@cs.virginia.edu

ABSTRACT

Source code modifications are often documented with log messages. Such messages are a key component of software maintenance: they can help developers validate changes, locate and triage defects, and understand modifications. However, this documentation can be burdensome to create and can be incomplete or inaccurate.

We present an automatic technique for synthesizing succinct human-readable documentation for arbitrary program differences. Our algorithm is based on a combination of symbolic execution and a novel approach to code summarization. The documentation it produces describes the *effect* of a change on the runtime behavior of a program, including the conditions under which program behavior changes and what the new behavior is.

We compare our documentation to 250 human-written log messages from 5 popular open source projects. Employing a human study, we find that our generated documentation is suitable for supplementing or replacing 89% of existing log messages that directly describe a code change.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

General Terms

documentation, human factors, measurement

Keywords

commit messages, code summarization, differencing

*This work was supported by, but does not reflect the views of: NSF CCF 0954024, NSF CCF 0916872, NSF CNS 0716478, AFOSR FA9550-07-1-0532, and Microsoft Research.

ASE '10

1. INTRODUCTION

Professional software developers spend most of their time trying to understand code [15, 26]. Moreover, maintenance can typically be expected to consume 70–90% of the total lifecycle budget of a software project [27, 31]. Maintaining and evolving high-quality documentation is crucial to help developers understand and modify code [12, 25].

In this paper, we focus on the documentation of code changes. Much of software engineering can be viewed as the application of a sequence of modifications to a code base. Modifications can be numerous; the linux kernel, which is generally considered to be "stable" changes 5.5 times per hour [14]. To help developers validate changes, triage and locate defects, and generally understand modifications version control systems permit the association of a free-form textual log message (or "commit message") with each change. The use of such messages is pervasive among development efforts with versioning systems [24]. Accurate and up-to-date commit messages are desired [30], but on average, for the benchmarks we investigate in this paper, only two-thirds of changes are documented with a commit message that actually describes the change.

The lack of high-quality documentation in practice is illustrated by the following indicative appeal from the `MacPorts` development mailing list: "Going forward, could I ask you to be more descriptive in your commit messages? Ideally you should state what you've changed and also why (unless it's obvious) ... I know you're busy and this takes more time, but it will help anyone who looks through the log ..."[1]

In practice, the goal of a log message may be to (A) summarize *what* happened in the change itself (e.g., "Replaced a warning with an IllegalArgumentException") and/or (B) place the change in context to explain *why* it was made (e.g., "Fixed Bug #14235"). We refer to these as WHAT and WHY information, respectively. While most WHY context information may be difficult to generate automatically, we hypothesize that it is possible to mechanically generate WHAT documentation suitable for replacing or supplementing most human-written summarizations of code change effects.

We observe that over 66% of log messages (250 of 375; random sample) from five large open source projects contain WHAT summarization information, implying that tools like `diff`, which compute the precise textual difference between two versions of a file, cannot replace human documentation effort — if `diff` were sufficient, no human-written summarizations would be necessary. We conjecture that there are

[1] `http://lists.macosforge.org/pipermail/macports-dev/2009-June/008881.html`

two reasons raw diffs are inadequate: they are too long, and too confusing. To make change descriptions clearer and more concise, we propose a semantically-rich summarization algorithm that describes the effects of a change on program behavior, rather than simply showing what text changed.

Our proposed algorithm, which we refer to as DeltaDoc, summarizes the runtime conditions necessary for control flow to reach the changed statements, the effect of the change on functional behavior and program state, and what the program used to do under those conditions. At a high level this produces structured, hierarchical documentation of the form:

```
When calling A(), If X, do Y Instead of Z.
```

Additionally, as a key component, we introduce a set of iterative transformations for brevity and readability. These transformations allow us to create explanations that are 80% shorter than standard diffs.

The primary goal of our approach is to reduce human effort in documenting program changes. Experiments indicate that DeltaDoc is suitable for replacing the What content in as much as 89% of existing version control log messages. In 23.8% of cases, we find DeltaDoc contains more information than existing log messages, typically because it is more precise or more accurate. Moreover, our tool can be used when human documentation is not available, including the case of machine-generated patches or repairs [37].

The main contributions of this paper are:

- An empirical, qualitative study of the use of version control log messages in several large open source software systems. The study analyzes 1000 messages, finding that their use is commonplace and that they are comprised of both What and Why documentation.
- An algorithm (DeltaDoc) for describing changes and the conditions under which they occur, coupled with a set of transformation heuristics for change summarization. Taken together, these techniques automatically generate human-readable descriptions of code changes.
- A novel process for objectively quantifying and comparing the information content of program documentation.
- A prototype implementation of the algorithm, and a comparison of its output to 250 human-written messages from five projects. Our experiments, backed by a human study, suggest DeltaDoc could replace over 89% of human-generated What log messages.

We begin with an example highlighting our approach.

2. MOTIVATING EXAMPLE

Projects with multiple developers need descriptive log messages, as this quote from the cayenne project's development list suggests: "Sorry to be a pain in the neck about this, but could we please use more descriptive commit messages? I do try to read the commit emails, but since the vast majority of comments are *CAY-XYZ*, I can't really tell what's going on unless I then look it up."[2] Consider Revision 3909 of iText, a PDF library written in Java. Its entire commit log message is "Changing the producer info." This documentation is indicative of many human-written log messages that do not describe changes in sufficient detail for others to fully understand them. For example, neither the type of "the producer" nor the nature of the change to the "info" are clear. Now consider the DeltaDoc for the same revision:

```
When calling PdfStamperImp close()
 If indexOf(Document.getProduct()) != -1
  set producer = producer +
   "; modified using " + Document.getVersion()
 Instead of
  set producer = Document.getVersion() +
  "(originally created with: " + producer + ")"
```

With this documentation it is possible to determine that the change influences the format by which version information is added to the producer String. Our documentation is designed to provide sufficient context for the effect of a change to be understood. Furthermore, we claim that diff, a commonly-used tool for viewing the text of code changes, often fails to clarify their effects. Consider revision 3066 to the jabref bibliography manager also written in Java. It consists of a modification to a single return statement, as shown in the diff summarization below:

```
19c19,22
<     else return "";
---
>     else return pageParts[0];
>     //else return "";
```

Although the source code diff shows exactly what changed textually, it does not reveal the impact of the change. For example, without understanding more about the variable pageParts, the new behavior remains unclear. While diff can be instructed to provide additional adjacent lines of context, or can be viewed though a specialized tool to increase available context, such blanket approaches can become increasingly verbose without providing relevant information — as in this example, where pageParts is defined six lines above the change, out of range of a standard three-line context diff. Similarly, the diff does not explain the conditions under which the changed statement will be executed. By contrast, DeltaDoc explains what the program does differently and when:

```
When calling LastPage format(String s)
  If s is null
    return ""
  If s is not null and
        s.split("[-]+").length != 2
    return s.split("[-]+")[0]
        Instead of return ""
```

Because it links behavioral changes to the conditions under which they occur, this message yields additional clues beyond the diff output. We claim that documentation of this form can provide significant insight into the true impact of a change without including verbose and extraneous information. That is, it can serve to document What a change does. We formalize this intuition by studying the current state of log messages and proposing an algorithm to generate such documentation.

3. EMPIRICAL STUDY OF LOG MESSAGES

In this section, we analyze the reality of commit log documentation for five open source projects. The benchmarks used are shown in Table 1; they cover a variety of application domains. From each project, we selected a range of

[2] http://osdir.com/ml/java.cayenne.devel/2006-10/msg00044.html

Name	Revisions	Domain	kLOC	Devs
FreeCol	2000–2200	Game	91	33
jFreeChart	1700–1900	Data Presenting	305	3
iText	3800–4000	PDF utility	200	14
Phex	3500–3700	File Sharing	177	13
Jabref	2500–2700	Bibliography	107	28
total	1000		880	91

Table 1: The set of benchmark programs used in this study. The "Revisions" column lists the span of commits to the program analyzed for the study. The "Devs" column counts the number of distinct developers (by user ID) that checked in modifications during the lifetime of the project.

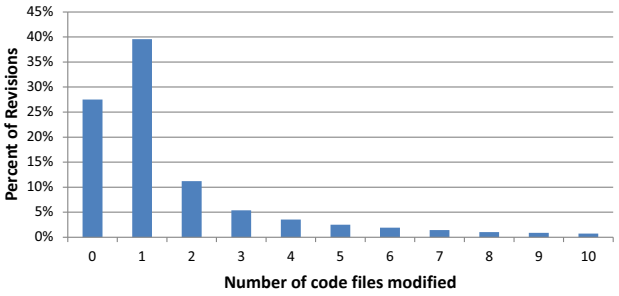

Figure 1: Percent of changes modifying a given number of source code files. This graph reports modifications to code files only: file additions or deletions, as well as modifications to non-code files, are excluded. By this metric, the majority of revisions (84%) involve changes to three or fewer source code files.

two-hundred revisions taking place after the earliest, most chaotic stages of development. The projects averaged 18 developers. While even single-author projects benefit from clear documentation, the case for such documentation is particularly compelling when it is a form of communication between individuals.

We first note that log messages are nearly universal. Across the 1000 revisions studied, 99.1% had non-empty log messages. Since such human-written documentation occurs so pervasively, we infer that it is a desired and important part of such software development. However, humans desire up-to-date, accurate documentation [25], and the varying quality of extant log messages (see Section 5.2) suggests that many commits could profit from additional explanation [5, 13].

The average size of non-empty human-written log messages is 1.1 lines. The average size of the textual `diff` showing the code change associated with a commit was 37.8 lines. Humans typically produce concise explanations or single-line references to bug database numbers. The longest human-written log-message was 7 lines. We hypothesize that messages of the same order of magnitude (i.e., 1–10 lines) would be in line with human standards, while longer explanations (e.g., 37-line `diffs`) are insufficiently concise. We use lines because they are a natural metric for size, but these findings also apply to raw character count.

We next address the scope of commits themselves. In Figure 1 we count the number of already-existing source code

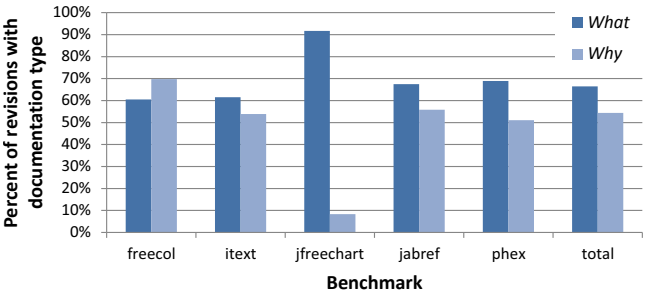

Figure 2: Percent of changes with WHAT and WHY documentation. In this study 375 indicative documented changes (about 75 from each benchmark; see Section 5.2.1 for methodology) were manually annotated as WHAT, WHY or both. WHAT documentation is 12% more common on average.

files that are modified in a given change. Note that this number may be zero if the change introduces new files, deletes old files, or changes non-code files (e.g., text or XML files, etc.) which our algorithm does not document. We note that the majority of changes edit one to three files, suggesting that a lightweight analysis that focuses on a small number of files at a time would be broadly applicable. These findings are similar to those of Purushothaman and Perry [28].

Finally, we characterize how often human-written log messages contain WHAT and WHY information. For this portion of the study, we selected a subset of 75 indicative log messages from each benchmark (see Section 5.2.1 for details) and manually annotated them as containing WHAT information, WHY information, or both. For this annotation, WHAT information expresses one or more specific relations between program objects (e.g., `x is null`). In Section 5.2.2 we describe the annotation process in detail. WHY information was defined to be all other text. We see that humans include both WHAT and WHY information with a slight preference for WHAT. The `freecol` project, an interactive game, is an outlier: its messages often reference game play. We conclude that a significant amount of developer effort is exercised in composing WHAT information.

Given this understanding of the human-written log messages, the next section describes our approach for producing moderately-sized and human-readable WHAT documentation for arbitrary code changes.

4. ALGORITHM DESCRIPTION

We propose an algorithm called DELTADOC that takes as input two versions of a program and outputs a human-readable textual description of the effect of the change between them on the runtime behavior of the program (i.e., the output is WHAT documentation). Our algorithm produces documentation describing changes to methods. It does not document methods that are created from nothing or entirely deleted. Instead, any such methods are simply listed in a separate step. Our algorithm is intraprocedural. If a change involves multiple methods or files we document changes to each method separately.

DELTADOC follows a pipeline architecture shown in Figure 3. In the first phase, as detailed in Section 4.1, we use symbolic execution to obtain path predicates for each statement in both versions of the code. In phase two, we identify

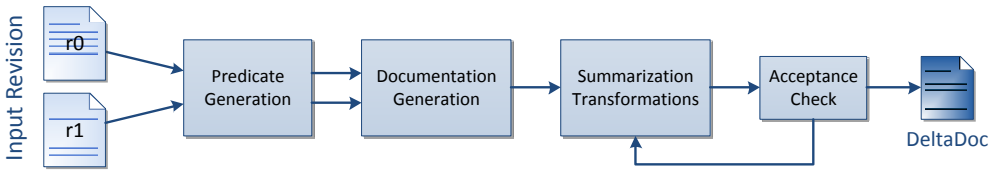

Figure 3: High-level view of the architecture of DeltaDoc. Path predicates are generated for each statement. Inserted statements, deleted statements, and statements with changed predicates are documented. The resulting documentation is subjected to a series of lossy summarization transformations until it is deemed acceptable (i.e., sufficiently concise).

and document statements which have been added, removed, or have a changed predicate (Section 4.2). Third, we iteratively apply lossy summarization transformations (Section 4.3) until the documentation is sufficiently concise. Finally, we group and print the statements in a structured, readable form.

4.1 Obtaining Path Predicates

Our first step is to obtain intraprocedural *path predicates*, formulae that describe conditions under which a path can be taken or a statement executed [2, 9, 17, 29]. To process a method, we first enumerate its loop-free control flow paths. We obtain loop-free paths by adding a statement to a path at most once: in effect, we consider each loop to either be taken once, or not at all. This decision can occasionally result in imprecise or incorrect documentation, however we show in Section 5.2 that this occurs infrequently in practice.

Each control flow path is symbolically executed (as in [5, 11]). We track conditional statements (e.g., `if` and `while`) and evaluate their guarding predicates using the current symbolic values for variables. When a statement has two successors, we duplicate the dataflow information and explore both: our analysis is thus path-sensitive and potentially exponential. We collect the resulting predicates; in conjunction, they form the path predicate for a given statement in the method. Some statements, such as the assignments to local variables, are heuristically less relevant to explaining What a change does than others, such as function calls or return statements. Our enumeration method is parametric with respect to a *Relevant* predicate that flags such statements. In our prototype implementation, invocation, assignment, `throw`, and `return` statements are deemed relevant. We symbolically execute all subexpressions of these statements and store the result — later documentation steps will use the symbolic (i.e., partially-evaluated) value instead of the raw textual value to describe a statement (i.e., E in Figure 4). An off-the-shelf path predicate analysis and symbolic execution analysis could also be used with relevancy filtering as a post-processing step; we combine the three for efficiency.

4.2 Generating Documentation

In this phase of the DeltaDoc algorithm, two sets of statement → predicate mappings, obtained from path predicate generation and representing the code before and after a revision, are used to make documentation of the form `If X, Do Y Instead of Z`.

Figure 4 shows the high-level pseudo-code for this phase. The algorithm matches statements with the same bytecode instruction and symbolic operands, enumerating all statements that have a changed predicate or that are only present in one version (lines 1–9). In our prototype implementation for Java programs, statements are considered to be unchanged if they execute the same byte code instruction under the same conditions.

Statements are then grouped by predicate (lines 10–14) and frequently-occurring predicates are handled first (line 15). Documentation is generated in a hierarchical manner. All of the statements guarded by a given predicate are sorted by line number (lines 17 and 21). Statements that are exactly guarded by the current predicate are documented via `Do` or `Instead of` phrases (lines 18 and 22) and removed from consideration. If multiple statements are guarded by the current predicate, they are printed in the same order they appear in the original source. If undocumented statements remain, the next most common predicate is selected (line 25) and documented using a `If` phrase (line 27). The helper function is then called recursively to document statements that might now be perfectly covered by the addition of the new predicate. The process terminates when all statements that must be documented have been documented; the process handles all such statements because the incoming set of interesting predicates is taken to be the union of all predicates guarding all such statements.

The statements and predicates themselves are rendered via a *Describe* function that pretty-prints the source language with minor natural language replacements (e.g., "is not" for `!=`). This algorithm produces complete documentation for all of its input (i.e., all heuristically *Relevant* statements see Section 4.3). For large changes, however, the documentation may be insufficiently concise; the next phase of the algorithm summarizes the result.

Finally, our algorithm also prints a list of fields or methods that have been added or removed. This mimics certain styles of human-written documentation, as in the following example from `jFreeChart` revision 1156:

```
(tickLength): New field,
(HighLowRenderer): Initialise tickLength,
(getTickLength): New method
```

4.3 Summarization Transformations

Summarization is a key component of our approach. Without explicit steps to reduce the size of the raw documentation created in Section 4.2, the output is likely too long and too confusing to be useful. We base this judgment on the observation that human-written log messages are on the order of 1–10 lines long (see Section 3). Without summarization, the raw output is, on average, twice as long as `diff`. In a previous study of human judgments of source code readability, the length of a snippet of code and the number of identifiers it contained were the two most relevant factors — and both were negatively correlated with readability [6].

Input: Method P_{old} : statements → path predicates.
Input: Method P_{new} : statements → path predicates.
Input: Mapping E : statements → symbolic statements.
Output: emitted structured documentation
Global: set *MustDoc* of statements = $\emptyset$

1: **let** *Inserted* = $\mathsf{Domain}(P_{new}) \setminus \mathsf{Domain}(P_{old})$
2: **let** *Deleted* = $\mathsf{Domain}(P_{old}) \setminus \mathsf{Domain}(P_{new})$
3: **let** *Changed* = $\emptyset$
4: **for all** statements $s \in \mathsf{Domain}(P_{new}) \cap \mathsf{Domain}(P_{old})$ **do**
5: **if** $P_{new}(s) \neq P_{old}(s)$ **then**
6: *Changed* ← *Changed* ∪ $\{s\}$
7: **end if**
8: **end for**
9: *MustDoc* ← *Inserted* ∪ *Deleted* ∪ *Changed*
10: **let** *Predicates* = $\emptyset$ (a multi-set)
11: **for all** statements $s \in$ *MustDoc* **do**
12: **let** $C_1 \wedge C_2 \cdots \wedge C_n = P_{new}(s) \wedge P_{old}(s)$
13: *Predicates* ← *Predicates* ∪ $\{C_1\} \cup \cdots \cup \{C_n\}$
14: **end for**
15: *Predicates* ← *Predicates* sorted by frequency
16: **call** HierarchicalDoc $(\emptyset, , P_{new}, P_{old}, Predicates, E)$

helper fun HierarchicalDoc$(p, P_{new}, P_{old}, Predicates, E)$
17: **for all** statements $s \in$ *MustDoc* with $P_{new}(s) = p$, sorted **do**
18: **output** "`Do` *Describe*$(E(s))$"
19: *MustDoc* ← *MustDoc* $\setminus \{s\}$
20: **end for**
21: **for all** statements $s \in$ *MustDoc* with $P_{old}(s) = p$, sorted **do**
22: **output** "`Instead of` *Describe*$(E(s))$"
23: *MustDoc* ← *MustDoc* $\setminus \{s\}$
24: **end for**
25: **for all** predicates $p' \in$ *Predicates*, in order **do**
26: **if** *MustDoc* ≠ $\emptyset$ **then**
27: **output** "`If` *Describe*(p)" and tab right
28: **call** HierarchicalDoc$(p \wedge p', P_{new}, P_{old}, Predicates$ with all occurrences of p' removed, $E)$
29: **output** tab left
30: **end if**
31: **end for**

Figure 4: High-level pseudo-code for Documentation Generation.

To mitigate the danger of unreadable documentation, we introduce a set of summarization transformations.

Our transformations are synergistic and can be sequentially applied, much like standard dataflow optimizations in a compiler. Just as copy propagation creates opportunities for dead code elimination when optimizing basic blocks, removing extraneous statements creates opportunities for combining predicates when optimizing structured documentation. Unlike standard compiler optimizations, however, not all of our transformations are semantics-preserving. Instead, many are *lossy*, sacrificing information content to save space. As the number of facts to document increases, so to does the number of transformations applied.

In the rest of this subsection, we detail these transformations. Due to space limitations, we use a number of example transformation templates; note however that the actual implementations are more robust. Where possible, we relate the transformations below to standard compiler optimizations that are similar in spirit; because our transformations are lossy and apply to documentation, this mapping is not precise.

Finally, note that our algorithm runs on control flow graphs, and is suitable for use with common imperative languages. Because our prototype implementation is for Java, we describe some aspects of the algorithm (particularly readability transformations in Section 4.3) in the context of Java. However, all of these transformations have natural analogs in most other languages.

Statement Filters. We document only a set of *Relevant* statements. Method invocations, which potentially represent many statements, are retained. We also retain assignments to fields, since they often capture the impact of code on visible program state. Finally, we document `return` and `throw` statements, which capture the normal and exceptional function behavior of the code. Notably, we avoid documenting assignments to local variables; our use of symbolic execution when documenting statements (as used in lines 18 and 22 of Figure 4) and predicates (line 27 of Figure 4) means that many references to local variables will be replaced by references to their values. We also take advantage of a standard Java textual idiom for accessor (i.e., "getter") methods: we do not document calls to methods of the form `get[Fieldname]()`, since they are typically equivalent to field reads.

Single Predicate transformations. Our algorithm produces hierarchical, structured documentation, where changes are guarded by predicates that describe when they occur at run-time. A group of associated statements may include redundant or extraneous terms. We thus consider all of the subexpressions in the statement group and:

1. Drop method calls already appearing in the predicate.
2. Drop method calls already appearing in a `return`, `throw`, or assignment statement.
3. Drop conditions that do not have at least one operand in documented statements.

Transformation 1 and 2 can be viewed as a lossy variant of common subexpression elimination: it is typically more important to know that a particular method was called than how many times it was called. Transformation 3 is based on the observation that predicates can become large and confusing when they contain many conditions that are not directly related to the statements they guard. We thus remove conditions that do not have operands or subexpressions that also occur in the statements they guard. For example:

```
If s != null and x is true,
   b is true and c is true, return s   →

If s != null, return s
```

because the guards discussing `a`, `b`, and `c` are unrelated to the action `return s`. However, this transformation is only applied if there are at least three guard conditions and at least one would remain; these heuristics work in practice to prevent too much context from being lost. Thus, we retain `If s is null, throw new Exception()` even though it does not share subexpressions with its statement.

Whole Change transformations reduce redundancy across the documentation of multiple changes to a single method.

They are conceptually akin to global common subexpression elimination or term rewriting. After generating the documentation for all changes to a method, we:

1. Inline `Instead of` conditions.
2. Inline `Instead of` predicates.
3. Add hierarchical structure to avoid duplication.

Transformation 1 converts an expression of the form:

```
If P, Do X Instead of If P, Do Y   →
If P, Do X Instead of Y
```

Similarly, transformation 2 converts:

```
If P, Do X Instead of if Q, Do X   →
Do X If P, Instead of If Q
```

The third transformation reduces predicate redundancy by nesting specific guards "underneath" general guards. Given two adjacent pieces of documentation, we merge them into a single piece of hierarchically-structured documentation by recursively finding and elevate the largest intersection of guard clauses they share. This is conceptually similar to conditional hoisting optimizations. For example, we convert:

```
If P and Q, Do X                    If P and Q,
If P and Q and R, Do Y      →         Do X
                                    If R,
                                      Do Y
```

Simplification is a lossy transformation that saves space by eliding the targets and arguments of method calls. For example, `obj.method(arg)` may be transformed to `obj.method()`, `method(arg)`, or simply `method()`.

In addition, documented predicates that are still greater than 100 characters are not displayed. This heuristic limit could ideally be replaced by a custom viewing environment where users could expand or collapse predicates as desired.

High-level summarization is applied only in the case of large changes, where the documentation is still too long to be considered acceptable (e.g., more than 10 lines). For sweeping changes to code, we observe that high-level trends are more important than statement-level details. We thus remove predicates entirely, replacing them with documentation listing the invocations, assignments, return values and throw values that have been added, removed or changed. We have observed that this information often is sufficient to convey what components of the system were affected by the change when it would be impractical to describe the change precisely.

This transformation is similar to high-level documentation summarization from the domain of natural language processing, in that it attempts to capture the gist of a change rather than its details.

Readability enhancements are always applied. We employ a small number of recursive pattern-matching translations to phrase common Java idioms more succinctly in natural language. For example:

- `true` is removed
- `x != 0` becomes "`x` is not null"
- `x instanceof T` becomes "`x` is a `T`"
- `x.hasNext()` becomes "`x` is nonempty"
- `x.iterator().next()` becomes "`x->`{some element}"

Similar readability transformations have proved effective in previous work to generate documentation for exceptions [5].

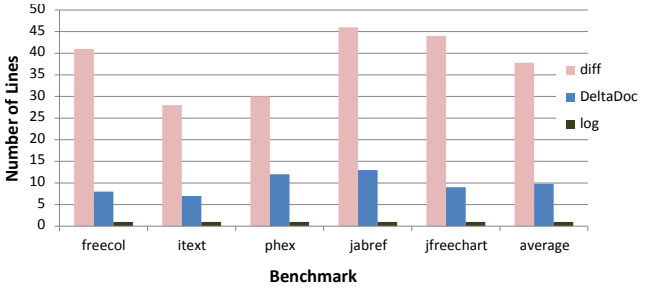

Figure 5: Size comparison between `diff`, DeltaDoc, and human generated documentation over all 1000 revisions from Table 1.

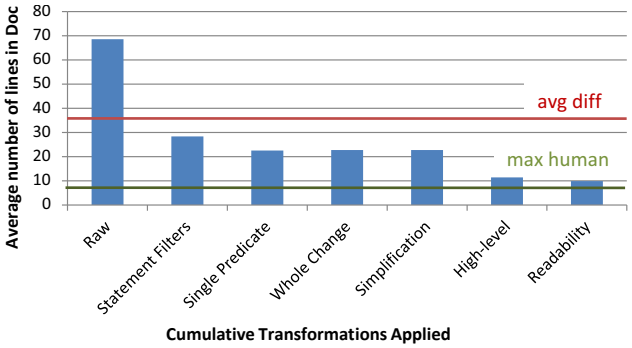

Figure 6: The cumulative effect of applying our set of transformations on the size of the documentation for all 1000 revisions from Table 1. We include the average size of `diff` output for comparison.

5. EVALUATION

Information and conciseness are important aspects of documentation [25]. A critical tradeoff exists between the information conveyed and the size of the description itself. In the extreme, printing the entire artifact to be documented is perfectly precise, but not concise at all. Conversely, empty documentation has the virtue of being concise, but conveys no information.

Our technique is designed to balance information and conciseness by imposing more summarization steps as the amount of information to document grows. In this evaluation, we aim to establish a tradeoff similar to that of real human developers.

We have implemented the algorithm described in Section 4 in a prototype tool for documenting changes to Java programs. In this section, we compare DeltaDoc documentation to human-written documentation and `diff`-generated documentation in terms of size (Section 5.1) and quality (Section 5.2). DeltaDoc takes about 1 second on average to document a change, and 3 seconds at most in our experiments. Documenting an entire repository of 10,000 revisions takes about 3 hours.

5.1 Size Evaluation

An average size comparison between baseline `diff` output, DeltaDoc, and target human documentation is presented in Figure 5. On average, DeltaDoc is approximately nine lines longer than the single-line documentation that humans typically create, though human-written documentation may be up to seven lines long. Notably, our documentation is

about 28 lines shorter than `diff` on average.

Figure 6 indicates how this level of conciseness is dependent upon the applications of the transformations described in Section 4.3. Our raw, unsummarized documentation is 68 lines long on average and approximately twice the size of `diff` output. Filtering statements reduces the size by 58%. Single predicate transformations restructure this documentation, removing significant redundancy and reducing the total size by another 21%. Whole change transformations are primarily for readability, and actually increase documentation size by about 1%. Simplifications have no effect on line count, but they do reduce the total number of characters displayed by 67%. High-level summarization, which is only applied to about 18% of changes, is effective at reducing the documentation size by another 49% overall. That is, high-level summarization is rarely applied, but when used it reduces overly-large documentation to a more manageable size. Finally, readability enhancements reduce the final line count by 13% (primarily by removing frequently-occurring `If true ...` predicates), however, they affect the total number of characters by less than 1%.

5.2 Content Evaluation

When comparing the quality of documentation for program changes, both the revisions selected for comparison and the criteria for quality are important considerations. We address each in turn before presenting our results.

5.2.1 Selection

Not all revisions to a source code repository can form the basis for a suitable comparison. For example, revisions to non-code files or with no human-written documentation do not admit an interesting comparison. We restricted attention to revisions that contained a non-empty log message with WHAT information, modified at least one and no more than three Java files, and modified non-test files (i.e., we ignore changes to unit test packages).

Revisions that touch one, two, or three Java files account for about 66–84% of all the changes that touch at least one file in our benchmarks (see Figure 1). We manually inspected revisions and log messages from each benchmark in Table 1 until we had obtained 50 conforming changes from each, or 250 total; this required the inspection of 375 revisions.

5.2.2 Documentation Comparison Rubric

We employed a three-step rubric for documentation comparison designed to be objective, repeatable, fine grained, and consistent with the goal of precisely capturing program behavior and structure. The artifacts to be compared are first cast into a formal language which expresses only specific information concerning program behavior and structure (steps 1 and 2). This transformation admits direct comparison of the information content of the artifacts (step 3).

Step 1: Each program object mentioned in each documentation instance is identified. For DELTADOC, this step is automatic. We inspect each log message for nouns that refer to a specific program *variable*, *method*, or *class*. This criterion admits only *implementation* details, but not *specification*, *requirement*, or other details. The associated source code is used to verify the presence of each object. Objects not named in the source code (e.g., "The User") are not considered program objects, and are not enumerated.

Type	Arity	Name
logical	unary	is *true* is *false*
	binary	or and equal to not equal to less-than greater-than less-than or equal to greater-than or equal to

programmatic	unary	is empty
		call / invoke
		`return`
		`throw`

	binary	assign to element of inserted in implies instance of
edit	unary	added changed removed

Table 2: Table of relational operators used for documentation comparison. *Logical* operators describe runtime conditions, *programmatic* operators express language concepts, and *edit* operators describe changes to code structure.

Step 2: A set of *true* relations among the objects identified in step 1 are extracted; this serves to encode each documentation into a formal language. Again, this process is automatic for DELTADOC. We inspect the human-written documentation to identify relations between the objects enumerated in step 1. Relations where one or both operands are implicit and unnamed are included. The full set of relations we consider are enumerated in Table 2, making this process essentially lock-step. Relations may be composed of constant expressions (e.g., 2), objects (e.g., X), mathematical expressions (e.g., $X + 2$), or nested relations. The associated source is used to check the validity of each encoded relation. If manual inspection reveals a relation to be false, it is removed. When in doubt, human-written messages were assumed to be true.

Step 3: For each relation in either the human documentation (H) or DELTADOC (D), we check to see if the relationship is also fully or partially present in the other documentation. For each pair of relations H and D, we determine if H *implies* D written $H \Rightarrow D$, or if $D \Rightarrow H$. A relation *implies* another relation if the operator is the same and each operand corresponds to either the same object, an object that is implicit and unnamed, or a more general description of the object (see below for an example of an unnamed object). If the relations are identical, then $H \Leftrightarrow D$. Again, the associated source code to check the validity of each relation. As with step 2, if the relation is found to be inconsistent with the program source, then it is removed from further consideration.

While steps 1 and 2 are lockstep, step 3 is not because it contains an instance of the *Noun phrase co-reference resolution problem*; it requires the annotator judge if two non-identical names refer to the same object (e.g., `getGold()` and `gold` both refer to the same program object). Automated approaches to this problem have been proposed (e.g., [23, 34]), however we use human annotators who have been shown to be accurate at this task [8].

This process is designed to distill from the message precisely the information we wish to model: the impact of the change on program behavior. Notably, we leave behind other (potentially valuable) information that is beyond the scope of our work (e.g., the impact of the change on performance). Furthermore, this process allows us to precisely quantify the difference in information content. This technique is similar to the NIST ROUGE metric [20] for evaluating machine-generated document summaries against human-written ones.

5.2.3 Score Metric

After applying the rubric, we quantify how well DeltaDoc captures the information in the log message with a metric we will refer to as the *score*. To compute the score we inspect each implication in the human-established mapping. For each $X \Rightarrow Y$, we add two points to the documentation containing X and one point to the documentation containing Y. This rewards more precise documentation (e.g., "updated version" vs. "changed version to 1.2.1"). For each $X \Leftrightarrow Y$ we add two points to each. For all remaining unmapped relations in the log message we add two points to the total for the log message; we do not for DeltaDoc, instead conservatively assuming any additional DeltaDoc relations are spurious. The score metric is computed as the fraction of points assigned to the algorithm documentation:

$$score = \frac{points\ for DeltaDoc}{points\ for DeltaDoc + points\ for\ log\ message}$$

The score metric ranges from 0 to 1, where 0.5 indicates the information content is the same. A score above 0.5 indicates that the DeltaDoc contains more information than the log message (i.e., there is at least one relation in the DeltaDoc not implied by the log message). The maximum score of 1.0 is possible only if the log message does not contain any correct information, but the DeltaDoc does. From an information standpoint, we conjecture that a score ≥ 0.5 implies that the DeltaDoc is a suitable replacement for human log message: it contains at least the same amount of What information. Figure 7 shows the average score we assigned to the documentation pairs from each benchmark.

After following the rubric, we asked 16 students from a graduate programming languages seminar to check our work by reviewing a random sample consisting of 32 documentation pairs. For each change, the annotators were shown the existing log message, the output of our algorithm, and the output of standard `diff`. The annotators were asked to inspect the information mapping we established and provide their own if they detected any errors. We then computed the score according to each annotator and compared the results to our original scores. Over all the annotators, we found the maximum score deviation to be 2.1% and the average deviation to be 0.48%. The pairwise Pearson interrater agreement is 0.94 [35].

The results of our study are presented in Figure 7. We

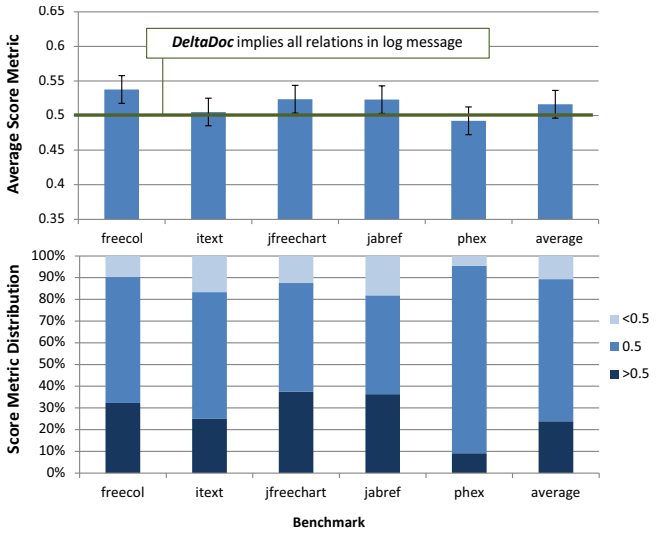

Figure 7: The distribution of *scores* for for 250 changes in 880kLOC that were already human-documented with What information. A score of 0.5 or greater indicates that the DeltaDoc is a suitable replacement for the log message. Error bars represent the maximum level of disagreement from human annotators.

compared the results on 250 revisions, 50 from each benchmark. Error bars indicate the maximum deviation observed from our human annotators and represent a confidence bound on our data. The average score for each benchmark except `phex` is greater than 0.5. `Phex`, however, shows the greatest number of changes with a score of at least 0.5. This is because `phex` contains a number of large changes that were not effectively documented by DeltaDoc according to our metric. These changes received a 0.0 and significantly reduced the average score. In general, our prototype DeltaDoc compares most favorably for small to medium sized changes because we impose tight space limitations. In practice, use of interactive visualization similar to hypertext allowing parts of DeltaDoc to be expanded or collapsed as needed would likely make such limitations unnecessary.

In 65.5% of cases we calculated the score to be 0.5; DeltaDoc contained the same information as the log message. For example, revision 3837 of `itext` contained the log message "no need to call clear()." The DeltaDoc contains the same relation resulting in a score of 0.5:

```
When calling PdfContentByte reset()
  No longer If stateList.isEmpty(),
    call stateList.clear()
```

In 23.8% of cases our documentation was found to contain more information (score> 0.5). In revision 2054 of `freecol`, the log message states "Commented unused constant." The DeltaDoc names the constant (score=0.67):

```
removed field: EuropePanel : int TITLE_FONT_SIZE
```

In the remaining 10.7% of cases, the DeltaDoc contained less information. Typically, this arises because of unresolved aliases or imprecision in either predicate generation or summarization. For example, the log message from revision 3111 of `jabref` states "Fixed bug: content selector for 'editor' field uses ',' instead of 'and' as delimiter." In this case the

DeltaDoc did not clearly contain a relation indicating ',' was removed while ' and ' was inserted:

```
When calling EntryEditor getExtra
 When ed.getFieldName().equals("editor")
  call contentSelectors.add(FieldContentSelector)
```

These findings support our hypothesis that the majority of What information in human-written log messages (about 89%) could be replaced with machine generated documentation without diminishing the amount of What information conveyed — reducing the overall effort required, and leaving humans with additional opportunities to focus on crafting Why documentation.

5.3 Qualitative Analysis

In addition to validation of our information mapping, for each documentation and log message pair, we asked our annotators which they felt "was more useful in describing the change." In 63% of cases the annotators either had no preference (17%) or preferred the DeltaDoc (46%).

We also asked a few free-form qualitative questions regarding our algorithm output. Many participants noted that the strength of the algorithm output is that it is "more specific" when compared to human-written log messages, but simultaneously "not as low level with no context" as compared to `diff` output. Fifteen of the sixteen participants agreed that the algorithm would provide a useful supplement to log messages. Many went further, noting that it is "very useful," "highly useful," "would be a great supplement," "definitely a useful supplement," and "can help make the logic clear." In comparison to human log messages the algorithm output was "often easier to understand," "more accurate ... easy to read," and "provides more information." The general consensus was that "having both is ideal," suggesting that DeltaDoc can serve as a useful supplement to human documentation when both are available.

5.4 Threats To Validity

Although our results suggest that our tool can efficiently produce documentation for commit messages that is as good as, or better than, human-written documentation, in terms of What information, they may not generalize.

First, the benchmarks we selected may not be indicative. We attempted to address this threat by selecting benchmarks from a wide variety of application domains (e.g., networking, games, business logic, etc.). Second, the revisions we selected for manual comparison may not be indicative. We attempted to mitigate this threat by selecting contiguous blocks of revisions from the middle of the projects' lifetimes, thus avoiding non-standard practices from the early phases of development. Bird *et al.* note that studies that attempt to tie developer check-ins to bug fixes via defect databases are subject to bias [4]; we mitigate this threat by inspecting all revisions, not just those tied to defect reports.

Second, our manual annotation of documentation quality may not be indicative. We attempted to mitigate this threat by clearly separating What and Why information in the evaluation and using an almost-mechanical list of criteria. With the subjective and context-dependent Why information removed, documentation can be evaluated solely on the fact-based What components. Furthermore, we asked 16 annotators to review a random sample of documentation pairs in order to quantify the uncertainty in our quality evaluation and mitigate the potential for bias: the error bars in Figure 7 indicate the maximum human disagreement.

6. RELATED WORK

Source code differencing is an active area of research. The goal of differencing is to precisely calculate which lines of code have changed between two versions of a program. Differencing tools are widely used in software engineering tasks including impact analysis, regression testing, and version control. Recently, Apiwattanapong *et al.* [1] presented a differencing tool that employs some semantic knowledge to capture object-oriented "correspondences." Our work is related to differencing, in that we also enumerate code changes. However, we recognize that the output of `diff` tools is often too verbose and difficult to interpret for use as documentation, since they necessarily list all changes. Our work is different because (1) our goal is to describe the impact of a change rather than the change itself (2) we focus on summarization instead of completeness and (3) our output is intended to be used as human-readable documentation and not as input to another analysis or tool.

Kim and Notkin described a tool called `LSDiff` which discovers and documents highlevel structure in code changes (e.g., refactorings) [18]. `LSDiff` has the potential to complement DeltaDoc which instead focuses on precise summarization of low-level differences.

Hoffman *et al.* [16] presented another approach to determining the difference between program versions. Like ours, their analysis focus on program paths. However, their tool is dynamic, and its primary application is regression testing and not documentation.

Automatic natural language document summarization was attempted as early as 1958 [21]. Varadarajan *et al.* [36] note that the majority of systems participating in the past *Document Understanding Conference* and the *Text Summarization Challenge* are extraction based. Extraction based systems extract parts of original documents to be used as summaries; they do not exploit the document structure. However, some analyses as early as Mathis *et al.* [22] in 1973 make significant use of structural information. Our analysis makes heavy use of the structure of our input since programming languages are much more structured than natural ones.

XML document summarization [10] and HTML summarization (for search engine preprocessing or web mining [7, 19, 33]) produce structured output. However, the intent in these cases is not to produce human-readable text, but to increase the speed of queries to the summarized data.

Finally, semantic differencing (e.g., [3, 32]) uses semantic cues in order to characterize the difference between two versions of a document. This characterization is often a binary judgment (e.g., "Is this change important?"), and is suitable for use in impact analysis and regression testing, but not for documentation.

7. CONCLUSION

Version control repositories are used pervasively in software engineering. Log messages are a key component of software maintenance, and can help developers to validate changes, triage and locate defects, and understand modifications. However, this documentation can be burdensome to create and can be incomplete or inaccurate, while undocumented source code `diff`s are typically lengthy and difficult to understand.

We propose DeltaDoc, an algorithm for synthesizing succinct, human-readable documentation for arbitrary program differences. Our technique is based on a combination of symbolic execution and a novel approach to code summarization. The technique describes what a code change does; it does not provide the context to explain why a code change was made. Our documentation describes the effect of a change on the runtime behavior of a program, including the conditions under which program behavior changes and what the new behavior is.

We evaluated our technique against 250 human-written log messages from five popular open source projects. We proposed a metric that quantifies the information relationship between DeltaDoc and human-written log messages. From an information standpoint, we found that DeltaDoc is suitable for replacing about 89% of log messages. A human study with sixteen annotators was used to validate our approach. This suggests that the "what was changed" portion of log message documentation can be produced or supplemented automatically, reducing human effort and freeing developers to concentrate on documenting "why it was changed."

8. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, 2007.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[3] D. Binkley, R. Capellini, R. Raszewski, and C. Smith. An implementation of and experiment with semantic differencing. In *International Conference on Software Maintenance*, page 82, 2001.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Foundations of Software Engineering*, pages 121–130, 2009.

[5] R. P. L. Buse and W. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[6] R. P. L. Buse and W. R. Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.

[7] D. Cai, X. He, J. Wen, and W. Ma. Block-level link analysis. *SIGIR Research and development in information retrieval*, pages 440–447, 2004.

[8] C. Cardie and K. Wagstaff. Noun phrase coreference as clustering. In *Joint Conference on Empirical Methods in NLP and Very Large Corpora*, pages 82–89, 1999.

[9] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[10] S. Comai, S. Marrara, and L. Tanca. XML document summarization: Using XQuery for synopsis creation. In *Database and Expert Systems Applications*, pages 928–932, 2004.

[11] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[12] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[13] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[14] A. M. Greg Kroah-Hartman, Jonathan Corbet. Linux kernel development. *The Linux Foundation*, 2009.

[15] P. Hallam. What do programmers really do anyway? In *Microsoft Developer Network (MSDN) — C# Compiler*, Jan 2006.

[16] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. *SIGPLAN Not.*, 44(6):453–464, 2009.

[17] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.

[18] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *International Conference on Software Engineering*, pages 309–319, 2009.

[19] C. Lee, M. Kan, and S. Lai. Stylistic and lexical cotraining for web block classification. In *Workshop on Web information and data management*, pages 136–143, 2004.

[20] C.-Y. Lin and F. J. Och. Looking for a few good metrics: Rouge and its evaluation. In *NTCIR Workshop*, 2004.

[21] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165, 1958.

[22] B. A. Mathis, J. E. Rush, and C. E. Young. Improvement of automatic abstracts by the use of structural analysis. *Journal of the American Society for Information Science*, 24(2):101–109, 1973.

[23] J. F. Mccarthy and W. G. Lehnert. Using decision trees for coreference resolution. In *Joint Conference on Artificial Intelligence*, pages 1050–1055, 1995.

[24] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, 2000.

[25] D. G. Novick and K. Ward. What users say they want in documentation. In *Conference on Design of Communication*, pages 84–91, 2006.

[26] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall, NJ, USA, 2001.

[27] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* John Wiley & Sons, Inc., 1996.

[28] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.

[29] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *International Conference on Software Engineering*, pages 478–488, 2002.

[30] M. J. Rochkind. The source code control system. *IEEE Trans. Software Eng.*, 1(4):364–370, 1975.

[31] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* Addison-Wesley Longman, MA, USA, 2003.

[32] E. Soechting, K. Dobolyi, and W. Weimer. Syntactic regression testing for tree-structured output. *International Symposium on Web Systems Evolution*, September 2009.

[33] R. Song, H. Liu, J. Wen, and W. Ma. Learning block importance models for web pages. In *International World Wide Web Conference*, pages 203–211, 2004.

[34] W. M. Soon, H. T. Ng, and D. C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Comput. Linguist.*, 27(4):521–544, 2001.

[35] S. E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9(4), 2004.

[36] R. Varadarajan and V. Hristidis. A system for query-specific document summarization. In *Information and knowledge management*, pages 622–631, 2006.

[37] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.