A Human-Centric Approach to Program Understanding

Ph.D. Dissertation Proposal
Raymond P.L. Buse
`buse@cs.virginia.edu`

April 6, 2010

1 Introduction

Software development is a large global industry, but software products continue to ship with known and unknown defects [60]. In the US, such defects cost firms many billions of dollars annually by compromising security, privacy, and functionality [73]. To mitigate this expense, recent research has focused on finding specific errors in code (e.g., [13, 25, 29, 34, 35, 47, 48, 61, 66, 86]). These important analyses hold out the possibility of identifying many types of implementation issues, but they fail to address a problem underlying all of them: software is difficult to understand.

Professional software developers spend over 75% of their time trying to understand code [45, 76]. Reading code is the most time consuming part [31, 39, 78, 85] of the most expensive activity [77, 87] in the software development process. Yet, software comprehension as an activity is poorly understood by both researchers and practitioners [74, 106].

Our research seeks to develop a general and practical approach for *analyzing* program understandability from the perspective of real humans. In addition, we propose to develop tools for mechanically *generating* documentation in order to make programs easier to understand. We will focus on three key dimensions of program understandability: **readability**, a local judgment of how easy code is to understand; **runtime behavior**, a characterization of what a program was designed to do; and **documentation**, non-code text that aids in program understanding.

Our key technical insight lies in combining multiple surface features (e.g., *identifier length* or *number of assignment statements*) to characterize aspects of programs that lack precise semantics. The use of lightweight features permits our techniques to scale to large programs and generalize across multiple application domains. Additionally, we will continue to pioneer techniques [19] for generating output that is directly comparable to real-world human-created documentation. This is useful for evaluation, but also suggests that our proposed tools could be readily integrated into current software engineering practice.

Software understandability becomes increasingly important as the number and size of software projects grow: as complexity increases, it becomes paramount to comprehend software and use it correctly. Fred Brooks once noted that "the most radical possible solution for constructing software is not to construct it at all" [16], and instead assemble already-constructed pieces. Code reuse and composition are becoming increasingly important: a recent study found that a set of programs was comprised of 32% re-used code (not including libraries) [88], whereas a similar 1987 study estimated the figure at only 5% [38]. In 2005, a NASA survey found that the most significant barrier to reuse is that software is too difficult to understand or is poorly documented [42] — above even requirements or compatibility. In a future where software engineering focus shifts from implementation to design and composition concerns, program understandability will become even more important.

2 Research Overview and Challenges

We propose to model aspects of program understandability and to generate documentation artifacts for the purposes of measuring and improving the quality of software. We couple programming language analysis techniques, such as dataflow analyses and symbolic execution, with statistical and machine learning techniques, such as regression and Bayesian inference, to form rich descriptive models of programs.

We believe that in addition to providing practical support for software development, descriptive models of program understandability may offer new and significant insight into the current state of large-scale software design and composition. We will create algorithms and models to analyze how code is written, how it is structured, and how it is documented. We will evaluate our models empirically, by measuring their accuracy the quality or behavior of software.

2.1 Measuring Code Readability

We define readability as a human judgment of how easy a text is to understand. In the software domain, this is a critical determining factor of quality [1]. The proposed research challenge is (1) to **develop a software readability metric** that agrees with human annotators as well as they agree with each other and scales to large programs. This analysis is based on textual code features that influence readability (e.g., indentation). Such a metric could help developers to write more readable software by quickly identifying code that scores poorly. It can assist in ensuring maintainability, portability, and reusability of the code. It can even assist code inspections by helping to focus effort on parts of a program that are mostly likely to need improvement. Finally, it can be used by other static analyses to rank warnings or otherwise focus developer attention on sections of the code that are less readable and, as we show empirically, more likely to contain bugs.

2.2 Predicting Runtime Behavior

Runtime Behavior refers to what a program is most likely to do — information that is typically unavailable for a static analysis. We claim that understanding runtime behavior is critical to understanding code. This conjecture is supported by the observation that runtime behavior information is a key aspect of documentation. First, consider that documentation is based on code summarization: if summarization were not important, then documentation would be unneeded as code would document itself. Second, summarization implicitly requires a prioritization of information based on factors including runtime behavior. For example, the function in Figure 1 from the Java standard library implementation of `Hashtable`, is documented by describing its expected most common behavior, "Maps the specified key to the specified value. . . " rather than describing what happens if `count >= threshold` or `value == null`. Our proposed technique identifies path features, such as "throws an exception" or "writes many class fields", that we find are indicative of runtime path frequency.

```
/**
* Maps the specified key to the specified
* value in this hashtable
*/
public void put(K key , V value)
{
   if ( value == null )
     throw new Exception();

   if ( count >= threshold )
     rehash();

   index = key.hashCode() % length;

   table[index] = new Entry(key, value);
   count++;
}
```

Figure 1: The `put` method from the Java SDK version 1.6's `java.util.Hashtable` class. Some code has been modified for illustrative simplicity.

The research challenge in this area is (2) to **statically predict the relative execution frequency** of paths through source code. This analysis is rooted in the way developers understand and write programs. If successful, we will be able to improve the utility of many profile-based hybrid analyses including optimizing compilers by reducing the need for profiling data, as well as to focus developer attention on corner cases while new program functionality is being created or existing programs are composed.

2.3 Generating Documentation

Our third area of investigation is program ***documentation***, which David Parnas claims is "the aspect of software engineering most neglected by both academic researchers and practitioners" [74]. We propose to develop tools and techniques to automatically generate human-readable documentation, thereby helping developers understand software. The remaining research challenges are (3) to **automatically generate documentation for exceptional situations** (4) to **automatically generate example documentation for APIs** and (5) to **automatically generate documentation for program changes**, patches and repairs. Challenges (3) and (4) would benefit software composition and integration. Challenge (5) would supplement or replace log messages in change management systems, complement self-certifying alerts [27], improve the usability of analyses that generate patches [100, 104], and aid defect triage [46, 51].

Exceptions: Modern exception handling allows an error detected in one part of a program to be handled elsewhere depending on the context [40]. This construct produces a non-sequential control flow that is simultaneously convenient and problematic [67, 82]. Uncaught exceptions and poor support for exception handling are reported as major obstacles for large-scale and mission-critical systems (e.g., [3, 17, 23, 89]). Some have argued that the best defense against this class of problem is the complete and correct documentation of exceptions [62].

Unfortunately, the difficulty of documenting exceptions leads to many examples like the one shown in Figure 2. Notice that the human provided JAVADOC documentation for the exception in this method is "`If the move is illegal`". This hides what constitutes an illegal move, which might be desirable if we expect that the implementation might change later. However, an automated tool could provide specific and useful documentation that would be easy to keep synchronized with an evolving code base. The algorithm we propose generates the additional JAVADOC "`IllegalStateException thrown when getLocation() is not a Europe.`"

```
/**
 * Moves this unit to america.
 *
 * @exception IllegalStateException
 *    If the move is illegal.
 */
public void moveToAmerica() {

  if (!(getLocation() instanceof Europe))
  {
    throw new IllegalStateException
      ("A unit can only be moved to" +
       " america from europe.");
  }

  setState(TO_AMERICA);
}
```

Figure 2: The function `Unit.moveToAmerica()` from `FreeCol`, an open-source game. Some code has been omitted for illustrative simplicity.

API Usage: In studies of developers, API usage examples have been found to be a key learning resource [50, 64, 72, 92, 96]. Conceptually, documenting how to *u*se an API is often preferable to simply documenting the function of each of its components.

Consider an example from Sun Microsystem's published documentation in Figure 3. This documentation demonstrates a common use case for `ObjectOutputStream`: supply a `FileOutputStream` to the constructor, write objects, and then call `close()`.

One study found that the greatest obstacle to learning an API in practice is "insufficient or inadequate examples" [81]. We propose to design an algorithm that automatically generates documentation of this type. Given a corpus of usage

```
FileOutputStream fos =
  new FileOutputStream("t.tmp");
ObjectOutputStream oos =
  new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
fos.close();
```

Figure 3: Excerpt from documentation of `java.util.ObjectOutputStream`.

examples (e.g., indicative uses mined from other programs), we propose to distill the most common use case and render it in a form suitable for use as documentation.

Changes: Much of software engineering can be viewed as the application of a sequence of modifications to a code base. Version control systems allow developers to associate a free-form textual log message (or "commit message") with each change they introduce [8]. The use of such commit messages is pervasive among development efforts that include version control systems [68]. Version control log messages can help developers validate changes, locate and triage defects, and generally understand modifications [7].

A typical log message describes either or both what was changed and why it was changed. For example, revision 2565 of Jabref, a popular bibliography system, is documented with "Added Polish as language option." Revision 3504 of Phex, a file sharing program, is documented with "providing a disallow all robots.txt on request." However, log messages are often less useful than this; they do not describe changes in sufficient detail for other developers to fully understand them. Consider Revision 3909 of `iText`, a PDF library: "Changing the producer info." Since there are many possible producers in `iText`, the comment fails to explain what is being changed, under what conditions, or for what purpose.

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] Our proposed technique is designed to replace or supplement human documentation of changes by describing 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.

3 Proposed Research

We propose several research thrusts for predicting and improving the quality of software as it is composed, evolved and maintained:

1. To **develop a software readability metric** that agrees with human judgments.
2. To statically **predict relative runtime execution frequencies**.
3. To **generate documentation for exceptional situations** automatically.
4. To **document APIs** with usage examples automatically.
5. To **document program changes** automatically.

In the rest of this section we describe the proposed research in detail: Section 3.1 describes our modeling approach and Section 3.2 describes output generation. In Section 4 we lay out our experimental design and evaluation for each research thrust.

3.1 Analysis and Characterization

The factors which contribute to program understanding are not well-known. We propose to use supervised learning to develop models of readability and runtime behavior based on a combination of easily-measured syntactic or semantic qualities of code.

We find machine learning to be particularly well-suited to exploring program understanding. First, because we have observed that many human opinions can be efficiently and accurately modeled based on simple "surface" qualities. These surface features are easy to extract, allowing our analyses to scale to large programs. Second, the models we propose are domain agnostic, making them robust in comparison to traditional approaches that rely on precisely defined specifications. For example, while work used to

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

software model check one device driver may be difficult to re-use when model checking a separate driver with a different correctness argument, a model of software readability or frequency generated from data on one project can generalize to other unrelated projects [20]. Our process also has the advantage that it can be applied incrementally, producing results immediately for local program modifications

We characterize software using a set of *features*. With respect to this feature set, each code artifact can be viewed as a vector of numeric entries: the *feature values* for that artifact. Additionally, for each learning task, we define a *class feature* (or *dependent feature*) that represents our learning goal (e.g., readability). Each model is trained on a set of example artifacts for which the class feature value is known. In the case of readability, human annotators provide the training data; to predict runtime behavior we can employ benchmark programs with indicative workloads. The model resulting from this training can be used on off-the-shelf software that lacks such labeled training data.

3.1.1 Research: Measuring Software Readability

Perhaps everyone who has written code has an intuitive notion of the concept of software readability, and that program features such as indentation (e.g., as in Python [99]), choice of identifier names [79], and comments play a significant part. Dijkstra, for example, claimed that the readability of a program depends largely upon the simplicity of its sequencing control, and employed that notion to help motivate his top-down approach to system design [32]. We propose to create descriptive and normative models of software readability based on these features, which can be extracted automatically from programs.

Human notions of code readability arise from a complex interaction of textual features (e.g., line length, whitespace, choice of identifier names, etc.). We propose to use a feature set based on this conjecture to model readability. We will employ human annotators to provide training input to the model. Such a model will allow us to characterize the extent to which humans agree on what makes code readable. It will also permit us to construct a metric for software readability suitable for use in software quality analysis and evaluation tasks. Our preliminary results, with data from computer science students, suggest that readability correlates with project stability and defect density [20].

3.1.2 Research: Predicting Runtime Behavior

We hypothesize that information about the expected runtime behavior of imperative programs, including their relative path execution frequency, is often embedded into source code by developers in an implicit but predictable way. We hypothesize that paths that change a large amount of program state (e.g., update many variables, throw an exception thus lose stack frames, etc.) are expected by programmers to be executed more rarely than paths that make small, incremental updates to program state (e.g., changes to a few fields).

We propose to construct a model of static program paths suitable for estimating runtime frequency. We consider method invocations between methods of the same class to be part of one uninterrupted path; this choice allows us to be very precise in common object-oriented code. Conversely, we split paths when control flow crosses a class boundary; this choice allows us to scale to large programs which we foresee as the principal challenge of our approach.

Central to our technique is *static path enumeration*, whereby we enumerate all acyclic intra-class paths in a target program. To train our model and experimentally validate our technique, we also require a process for *dynamic path enumeration* that counts the number of times each path in a program is executed during an actual program run. The goal of this work is to produce a static model of dynamic execution frequency that agrees with the actual execution frequency observed on held-out indicative workloads.

3.2 Documentation and Summarization

In addition to measuring code understandability, we propose to improve it by creating tools capable of automatically generating documentation. Our general strategy is to mimic human documentation by *summarizing the effect* of a statement on the functional behavior and state of a program, rather than simply printing the statement in question. Automatic documentation of this type holds out the promise of replacing much of human documentation effort with documentation artifacts that are more consistent, more accurate, and easier to maintain than traditional documentation.

Our technique is based on a combination of program analyses that can be used to summarize and simplify code. Symbolic execution [70] and dataflow analysis [80] allows us to condense sequences of instructions into single ones by evaluating and combining terms. We couple this with alias analysis (e.g., [28]) to resolve invocation targets. Additionally, we propose to develop new dynamic summarization heuristics to condense the representation of the documentation as needed. For path predicates, we can re-arrange terms to condense their textual representation. For statements, we can select for relevancy and importance especially in consideration of runtime behavior metrics (e.g., prefer to document a method call to an assignment statement).

We now describe three research thrusts based on the generation of documentation.

3.2.1 Research: Documentation of Exceptional Situations

We propose to construct an algorithm that statically infers and characterizes exception-causing conditions in programs. The output of the algorithm must be usable as human-readable documentation of exceptional conditions.

First, we locate exception-throwing instructions and track the flow of exceptions through the program. We propose two improvements over previous work [83]: an initial analysis and processing of the call graph for increased speed, and a more precise treatment of exception raising statements to ensure soundness.

Second, we symbolically execute control flow paths that lead to these exceptions. This symbolic execution generates boolean formulae over program variables that describe feasible paths. If the formula is satisfied at the time the method is invoked, then the exception can be raised. We then output a string representation of the logical disjunction of these formula for each method/exception pair. This step includes a set of transformations to increase readability and brevity of the final documentation.

A key challenge is ensuring scalability in a fully inter-procedural analysis. In addition to call-graph preprocessing and infeasible path elimination, we propose to enumerate paths "backward" from exception throwing statements. Because these statements are relatively rare (as compared to method call sites) this allows us to significantly reduce time needed for path enumeration.

3.2.2 Research: Documenting APIs with Usage Examples

We propose to use techniques from specification [59, 101, 103] and frequent itemset [2] mining to model API usage as sequences of method calls and construct documentation based on that model.

Given a target class or set of related methods to document, and a corpus of code that uses the methods, we construct a usage model: a set of finite state machines abstracting actual usage (as in [105]) along with a count of the number of occurrences conforming to each. To construct the usage model, we first statically enumerate intra-procedural static traces through the corpus. These traces are then sliced [52] to represent only uses of the class to be documented. That is, while many programs may use `socket`s, we wish to abstract away program-specific details (e.g., the data sent over the socket) while retaining API-relevant details (e.g., `open` before `read` or `write` before `close`). We then derive state machine models by traversing and symbolically executing the traces. Next, we merge the machines to minimize the set of patterns. Finally, the machines can be sorted by commonality and the model can be distilled as documentation.

3.2.3 Research: Documenting Program Changes

We propose to investigate how code changes are documented and seek to automate the process. Our goal is a precise characterization of the effect of a source code change on a program's execution.

In practice, the goal of a log message may be to (A) summarize *what* happened in the change itself (e.g., "Replaced a warning in Europe.add 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,

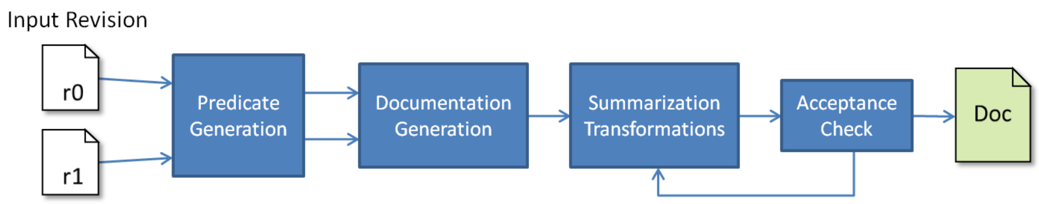

Figure 4: High-level view of the architecture of our algorithm for generating change documentation. Path predicates are generated for each statement. Inserted statements, deleted statements, and statements with changed predicates are documented. The resulting documentation is subjected to a series of lossy summarization transformations, as needed, until it is deemed acceptable (i.e., sufficiently concise).

we hypothesize that it is possible to mechanically generate WHAT documentation suitable for replacing most human-written summarizations of code change effects.

Our proposed algorithm summarizes the runtime conditions necessary to cause control flow to reach the changed statements, the effect of the changed statements on functional behavior and program state, as well as what the program used to do under those conditions. At a high level, our approach generates structured, hierarchical documentation of the form:

```
When calling A(), if X, do Y instead of Z.
```

Our proposed algorithm follows a pipeline architecture, as shown in Figure 4. In the first phase, we use symbolic execution to obtain *path predicates*, formulae that describe conditions under which a path can be taken or a statement executed [11, 24, 53, 84], for each statement in both versions of the code. In phase two, we identify statements which have been added, removed, or changed. Statements are grouped by predicate and documentation is generated in a hierarchical manner. All of the statements guarded by a given predicate are sorted by line number.

Summarization is a key component of our algorithm. Without explicit steps to reduce the size of the raw documentation we generate, 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 generally less than 10 lines long. To mitigate the danger of producing unreadable documentation, we introduce a set of transformations that can be sequentially applied.

Our summarization transformations are synergistic and can be iteratively 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. The goal of each transformation is to reduce the total amount of text in the final output while maintaining information value.

4 Proposed Experiments

In this section, we outline our proposed experimental methodology for each research thrust. We propose to gather data from and test on large, popular, open source programs. Accuracy and scalability are overarching concerns.

Each major thrust of our proposal can be investigated independently. However, we anticipate that success in one area can be extended to improve other analyses. For example, an accurate execution frequency metric can help us better characterize the impact of a program change and thus lead to more successful documentation generation.

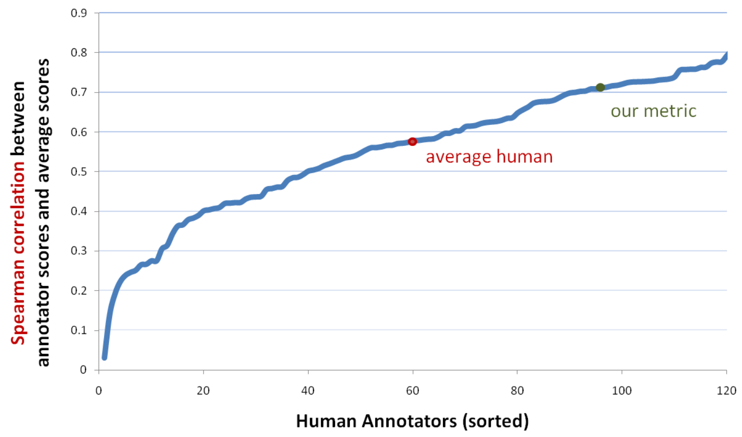

Figure 5: Inter-annotator agreement. For each annotator (in sorted order), the value of Spearman's ρ between the annotator's judgments and the average judgments of all the annotators (the model that we attempt to predict). Also plotted is Spearman's ρ for our metric as compared to the average of the annotators.

4.1 Experiments: Measuring Software Readability

Our metric for software readability is a mapping from an arbitrary source code snippet to a finite score domain (i.e., the closed interval [0,1]) indicating how easy the code is for a human reader to understand.

Our readability model can be evaluated directly, as a predictive model for human readability annotations, and indirectly by exploring correlations with traditional metrics for software quality. In our preliminary work [20] (Distinguished Paper Award, ISSTA 2008), we showed how a model derived from the judgments of 120 students, and based on a simple set of local code features, can be 80% accurate, and better than a human on average, at predicting readability judgments, as shown in Figure 5. Furthermore, we have demonstrated that this metric correlates strongly with two traditional measures of software quality, code changes and defect reports. For example, we have shown how our metric for readability correlates with the self-reported maturity of several large projects from `Sourceforge`, a repository for open-source software.

Our success in developing a normative model hinges on our human-based study of the readability of program changes. Our goal is to construct a model that agrees with human-judgments at the level at which study participants agree with each other. We use Spearman's ρ to measure correlation between judgements.

4.2 Experiments: Predicting Execution Behavior

Our model for execution frequency is a mapping from static, acyclic, intra-class program paths to a frequency estimate on the closed interval [0,1]. We propose to evaluate our model of execution frequency as a "temporal coverage metric" (analogous to a code coverage metric evaluation for test-cases) and as a static branch predictor.

For temporal coverage, we use our algorithm to enumerate and rank all static paths in the program. We then take the top k paths and run an instrumented version of the program on its indicative workloads, measuring how much time it spends on those k paths and how much time it does not. We deem our model successful if a small number of paths (i.e., small values of k) can explain more than half of run-time behavior. In preliminary work [18], we show that the predicted top 5% of program paths represent over 50% of the total runtime on the SPECjvm98 benchmarks. In addition, we can select one hot path per method and account for over 90% of the total runtime, even though the average number of paths per method is over 12 (see Figure 6).

We also propose to evaluate our model as a static branch predictor, thus admitting comparison with previous work in static characterization of runtime behavior. In such an experiment, the program is run

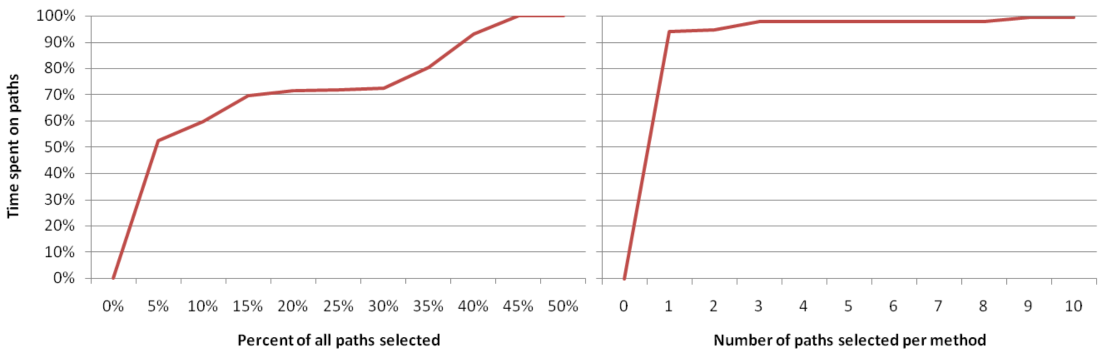

Figure 6: Percentage of the total run time covered by examining the top paths of the entire program and by examining the top paths in each method.

on an indicative workload and, at each branch, the static branch predictor is queried for a prediction. The fraction of successful predictions is the hit rate. A preliminary static branch predictor based on our technique has a 69% hit rate [18] compared to the 65% of previous work [10]. We consider our model successful if it can be used to predict branching behavior more accurately, on average, than the best approach currently known.

4.3 Evaluating Documentation

Each type of documentation artifact we propose to generate is designed to mimic a human-created documentation type found commonly in open source software. This property allows us to evaluate our generated documentation by direct comparison to human documentation — either manually or via a human study.

To evaluate each proposed tool, we will run it on a set of benchmark programs that contain the type of documentation in question. We will then pair each existing piece of documentation with the documentation suggested by our algorithm and manually review each pair. We propose to categorize each according whether the tool-generated documentation was *better*, *worse*, or about the *same* as the human-created version present in the code. In general, to be the same the generated documentation must contain all of the information in the matching human documentation. We consider it better only if the human documentation is clearly inaccurate, and worse in all other cases. To mitigate bias, the annotation will be precisely defined and carried out by multiple annotators. We can then utilize correlation statistics including Kohen's κ and Spearman's ρ to judge the degree of inter-annotator agreement. Documentation size will be considered separately from precision.

Additionally, we propose several user studies measuring the impact of our tools on human judgments and in simulated software engineering tasks. We have previous success with similar experiments involving human subjects [20].

We now briefly describe the experiment details for each documentation type and present examples. Additionally, we describe secondary experiments designed to measure, for example, how often the tools can be applied in real world software.

4.4 Experiments: Documentation of Exceptional Situations

For exception documentation, we plan to compare directly with embedded JAVADOC comments. Our preliminary work [19] involved almost two million lines of code and a direct comparison with 951 pieces of human-written documentation. Documentation generated by our prototype tool was at least as good or better 88% of the time and strictly better about 25% of the time.

Machine-generated documentation can be more accurate and much more complete than human documentation. For example, consider the following documentations proposed for a certain exception instance; first is the existing human-written documentation and second is a documentation generated by our algorithm:

```
 Existing:      id == null
Generated:      id is null or id.equals("")
```

We have also found that human-written prose is typically not necessary to succinctly convey information about program concepts such as exceptional behavior. In the following example, we observe that with properly named identifiers, documentation can be expressed clearly in programmatic terms.

```
 Existing:      has an insufficient amount of gold.
Generated:      getPriceForBuilding() > getOwner().getGold()
```

Perhaps most importantly, we found that humans only bothered to document legitimate exceptional situations 40% of the time. An automatic tool that generates candidate documentation would thus be of great help to ensure completeness of documentation for exceptions.

We will be successful if our machine-generated documentation is judged the same or better than human documentation at least 80% of the time in a wide range of open source programs. Our preliminary studies show this is both achievable and likely sufficient to promote adoption of the tool [19].

4.5 Experiments: Documentation of API Usage

For API usage documentation, we propose to compare the output of our algorithm to usage examples embedded in Javadoc and mined from the publicly-available sources, such as popular programs or reference manuals. To be successful, our generated examples must contain precisely the same information as in human written examples. For example, this machine-generated example describes a common use of `ObjectOutputStream` similarly to Figure 3 presented earlier.

```
FileOutputStream x = new FileOutputStream(string);
ObjectOutputStream y = new ObjectOutputStream(x);
y.write*(); // any number of times
y.close();
x.close();
```

Additionally, we propose a human study to directly evaluate the utility of the proposed tool. We will charge a group of upper-level computer science students with a few small software engineering tasks. Each task will require the use of an API that the students are not previously familiar with. Half of the students, the treatment group, will be allowed to view documentation generated by our tool. The other half, the control group, will only be presented with the existing documentation, which will not include any usage examples. We will then measure the time to competition and correctness of the student solutions. We hypothesize that the treatment group will complete the task more quickly and with fewer errors than the control group.

To evaluate the potential impact of our tool, we will also investigate how often classes contain methods that are good candidates for this type of documentation (i.e., methods that display a common use pattern).

We will be successful if (A) our machine-generated documentation is judged the same or better than human documentation at least 80% of the time, and (B) our human study shows a significant correlation between the speed and accuracy of participants and the presence of our tool-generated documentation.

4.6 Experiments: Documenting Program Changes

Our primary evaluation for program change documentation will be a manual comparison to version control log messages. For example, consider revision 3837 from `iText`, a PDF library. Aside from its length, we judge the generate output to be on par with the existing human output in this case, because both contain the same information, that `clear()` is no longer called.

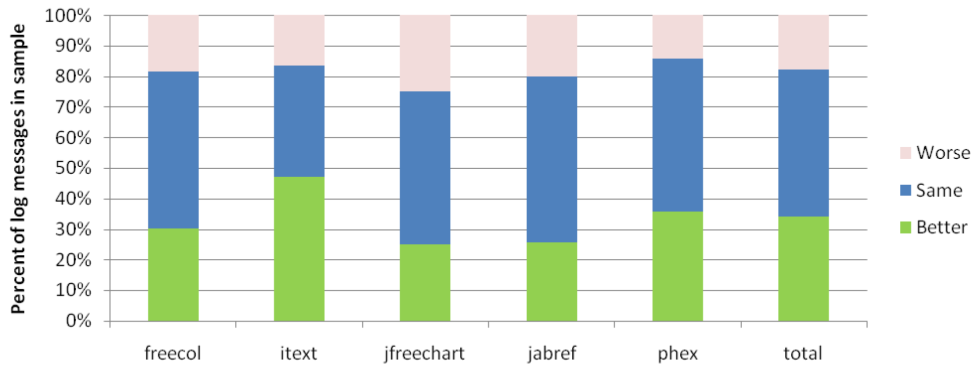

Figure 7: The relative quality of the at-most-10-line documentation produced by our prototype tool for 250 changes (50 per program) in 880kLOC that were already human-documented with WHAT information.

```
Existing:
  "no need to call clear()"
Generated:
  When calling PdfContentByte.reset()
    No longer if stateList.isEmpty(),
      call stateList.clear()
```

We have conducted an empirical study of 1000 commit messages. Based on its results, we have chosen “10 lines” as a maximal size cutoff: beyond that point, generated documentation is deemed insufficiently concise (by contrast, the average `diff` is typically over 35 lines long). Given a size cutoff, we will take code changes for which human-written log messages are available, generate documentation for them, and compare the WHAT information content of our generated messages to the human ones.

Figure 7 shows the example of such a comparison on 250 pieces of documentation generated by our prototype tool. To perform the evaluation, we first extracted all WHAT information from the human-written documentation: the generated documentation must include all true facts mentioned by the humans to be judged “same”. If the generated documentation is more accurate or more precise, it is judged “better”. All other cases are judged “worse”.

In addition to a manual assessment, we propose a study involving human subjects, who, unfamiliar with the program at hand are shown the patch (and the before and after code, if they desire) as well as the two documentations and asked to make a judgment. We will consider our algorithm successful if tool-generated patch documentation is judged by humans to be at least as accurate and useful as human-provided documentation in 80% of cases.

Finally, we will evaluate the size of our generated documentation, in lines and number of characters. This must be similar to the size of comparable human-created documentation to validate our summarization techniques.

5 Background

Readability. To further understand why an empirical and objective model of *software* readability is useful, consider the use of readability metrics in natural languages. The Flesch-Kincaid Grade Level [36], the Gunning-Fog Index [43], the SMOG Index [63], and the Automated Readability Index [55] are just a few examples of readability metrics that were developed for ordinary text. Despite their relative simplicity, they have each been shown to be quite useful in practice. Flesch-Kincaid, which has been in use for over 50 years, has not only been integrated into such popular text editors as Microsoft Word, but has also become a United States governmental standard. Agencies, including the Department of Defense, require many documents and forms, internal and external, to meet have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). Defense contractors also are often required to use it when they write technical manuals.

Much of the work in the area of source code readability today is based on coding standards (e.g., [5, 22, 97]). Style checkers such as PMD [26] and *The Java Coding Standard Checker* are employed to automatically enforce these standards. We propose to address two primary problems with these standards. First, they are designed largely arbitrarily; it is not known whether the practice they advocate is best (i.e., correlates with human judgments of readability). Second, their correlation with defect density and stable software evolution is unknown. We will develop a model of readability that correlates with human judgments as well as with conventional software quality metrics.

Current strategies for improving the readability and consistency of software and documentation include naming standards [94], software inspections [56], and analysis tools [6]. We find naming standards to be also largely arbitrary and not strongly correlated with code quality [20], while software inspections represent a large burden that may provide inconsistent results.

Runtime Behavior. The work most similar to our proposed approach is *static branch prediction*, first explored by Ball and Larus [10]. They showed that simple heuristics are useful for predicting the frequency at which branches are taken. Extensions to their work have achieved modest performance improvements by employing a larger feature set and neural network learning [21].

We have chosen to focus on predicting paths instead of predicting branches for two reasons. First, we claim that path-based frequency can be more useful in certain static analysis tasks, including our proposed work in machine generated documentation; see Ball *et al.* [12] for an in-depth discussion. Second, paths contain much more information than do individual branches, and thus are much more amenable to the process of formal modeling and prediction.

Automatic *workload generation* deals with constructing reasonable program inputs. This general technique has been adapted to many domains [58, 65, 69]. While workload generation is useful for stress testing software and for achieving high path coverage, it has not been shown suitable for the creation of indicative workloads that correlate strongly with standard test suites. In contrast, we attempt to model indicative execution frequencies.

More recent work in *concolic testing* [90, 91] explores all execution paths of a program with systematically selected inputs to find subtle bugs. It interleaves static symbolic execution to generate test inputs and concrete execution on those inputs. Symbolic values that are too complex for the static analysis to handle are replaced by concrete values from the execution. We propose to make static predictions about dynamic path execution frequency; we do not focus on or propose direct bug-finding.

Documentation. We propose to automatically generate human-readable descriptive documentation from unannotated source code. Many tools such as Doxygen, NDoc, Javadoc, Sandcastle, ROBODoc, POD, or Universal Report can be used to extract certain comments and software contracts from source code and create reference documents in such forms as text or HTML. While such tools can be useful by allowing for quick reference to developer-written comments, that utility is necessarily dependent on the conscientiousness of developers in providing comments that are accurate, complete, and up-to-date [71].

In practice, documentation and similar artifacts exhibit inconsistent quality, are frequently incomplete or unavailable, and can be difficult to measure in a meaningful way [19, 37, 49, 98]. One of the main difficulties in software maintenance is a lack of up-to-date documentation [30]. We are unaware of any previous effort to create machine-generated documentation suitable for supplementing or replacing these comments. We propose to reduce the documentation burden on developers by creating tools that produce three types of documentation automatically: for exceptions, for API usage, and for program changes.

API usage documentation is related to the well-researched topic of specification mining. A specification miner attempts to derive machine-checkable specifications by examining program source, traces, etc. Here, *specification* is loosely defined as "some formal description of correct program behavior" and is typically taken to mean "partial-correctness temporal safety property". Many miners encode legal program behavior specifications as finite state machines capturing such temporal safety properties [4, 59, 103, 105]. For example, a specification miner might identify that usages of `Stream.open()` must always be followed by `Stream.close()`. An important problem in specification mining is a high rate of false positives, defined as a candidate specification emitted by a miner that does not encode required behavior [34]. An example false positive is that `Iterator.hasNext()` must be followed by `Iterator.next()`: this behavior is common, but not required. Our proposed research avoids this pitfall and in fact takes advantage of it by mining common behavior. This observation allows us to leverage much previous work in modeling temporal properties of programs toward a new target: documentation.

Textual differencing utilities such as `diff`, represents the state-of-the-art in automatic change documentation. However, `diff` cannot replace human documentation effort. We conjecture that there are two reasons raw `diffs` are inadequate: they are too long and too confusing. We propose to overcome these issues by producing structured documentation describing the affect of a program change and by applying unique and empirically-validated summarization transformations.

6 Research Impact

A principal strength of our proposed research is generality. The work we propose can be easily adapted across languages, domains, and applications. We envision a series of models and tools that can be used to address many challenges associated with program understanding.

Improving Existing Analyses. Recent work has demonstrated that our preliminary execution path metric is useful for reducing false positives in the domain of specification mining [59], where it was three times as significant as weighting by code churn, path density or the presence of duplicate code and helped to reduce false positives by an order of magnitude.

Profile-guided optimization refers to the practice of optimizing a compiled binary subsequent to observing its runtime behavior on some workload (e.g., [9, 15, 44]). Our technique for runtime behavior prediction has the potential to make such classes of optimization more accessible; first, by eliminating the need for workloads, and second by removing the time required to run them and record profile information. A static model of relative path frequency could help make profile-guided compiler optimizations more mainstream.

Finite maintenance resources inhibit developers from inspecting and potentially addressing every warning issued by existing software analyses [57] or flagged in bug tracking systems [46, 51]. An analysis that produces a bug report that is never fixed is not as useful as it could be. Existing analysis reports, such as backtraces, are not a panacea: “there is significant room for improving users’ experiences ... an error trace can be very lengthy and only indicates the symptom ... users may have to spend considerable time inspecting an error trace to understand the cause.” [41] Our proposed metrics will help developers prioritize effort to bugs that are most likely to be encountered in practice. Furthermore, there has been a recent effort to create self-certifying alerts [27] and automatic patches [100, 104, 75, 93] to fix the flaws detected by static analyses. In mission-critical settings, such patches must still be manually inspected. Our proposed documentation tools will make it easier to read and understand machine-generated patches, increasing the trust in them and reducing the time and effort required to verify and apply them.

Evolution and Composition. Many problems arise from misunderstandings rather than classic coding errors [33]. Vulnerabilities may be introduced when code is integrated with third-party libraries or “injected” as a program is changed or maintained. The Spring Framework, a popular J2EE web application framework with over five million downloads [95], was discovered in September 2008 to have two such security vulnerabilities allowing attackers to access arbitrary files on the affected webserver [14]. Dinis Cruz, who helped identify the vulnerabilities, notes that they are “not security flaws within the Framework, but are design issues that if not implemented properly expose business critical applications to malicious attacks” [14]. Specifically, the vulnerability occurs when developers mistakenly assume that certain types of input fields are properly validated within the framework when they are not. Cruz notes that “developers don’t fully understand what’s happening. They don’t see the side effects of what they’re doing. In a way the framework almost pushes you to implement it in an insecure way” [14]. Malicious adversaries might also insert trojans or backdoors into open-source software, and outsourced or offshored components may not meet local standards. Automatically generated, up-to-date documentation would make it easier for integrators to avoid such problems by understanding the code [54]. Formal readability models and static path frequency predictions could focus

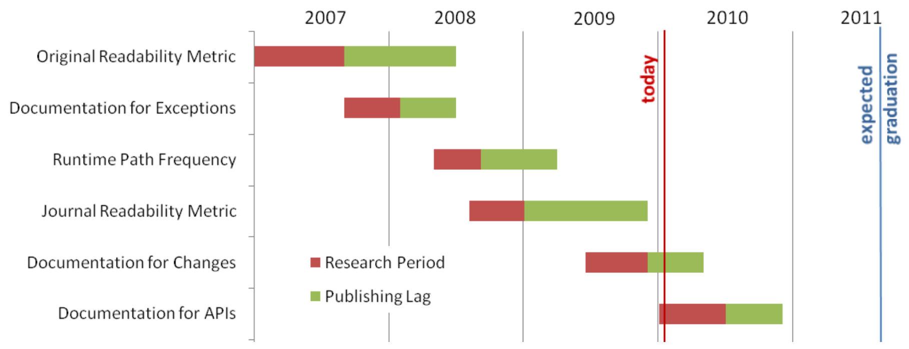

Figure 8: Proposed publication schedule.

attention on harder-to-understand code and corner cases — areas more likely to contain defects [20, 102].

Metrics. There are several popular readability metrics targeted at natural language [36, 43, 55, 63]. These metrics, while far from perfect, have become ubiquitous because they can very cheaply help organizations gain confidence that their documents meet goals for readability. We propose to formulate software readability metrics, backed with empirical evidence for effectiveness, to serve an analogous purpose in the software domain. For example, when constructing software from off-the-shelf components or subcontractors, the subcomponents might be required to meet a overall code readability standards to help ensure proper integration and safe modification and evolution in the future, just as they are required to meet English readability standards for documentation.

Identifying Expectation Mismatch. While our proposed metrics are designed to predict human judgments, in practice they will not always match expectations. Cases where the mismatch is the greatest may reveal code that should be scrutinized. A mismatch occurs when, for example, dynamic profiling reveals a path that was labeled uncommon by our metric but proves to be very common in practice. Such a situation may indicate a performance bug, where usage deviates from developer expectations. Similarly, our readability metric can help developers identify code that may not be as readable to others as they thought.

7 Research Plan

The proposed dissertation is comprised of three high-level research thrusts: readability, runtime behavior, and documentation. Furthermore, under that umbrella, we have detailed five specific projects which we propose to publish in six papers, three of which are now published (*A Metric for Software Readability* [20], *The road not taken: Estimating path execution frequency statically* [18], and *Automatic documentation inference for exceptions* [19]) and three of which are ongoing (*Learning a Metric for Code Readability*, *Automatic Documentation of Program Changes*, and *Automatic Documentation of APIs with Examples*). In the past we have targeted *The International Symposium on Software Testing and Analysis* (ISSTA) and *The International Conference on Software Engineering* (ICSE). In the future we intend to target these as well as *ACM Conference on Programming Language Design and Implementation* (PLDI), *The International Conference on Automated Software Engineering* (ASE), and the journal *IEEE Transactions on Software Engineering* (TSE).

Uncertainty in the schedule includes a possible industry internship during the summer of 2010 for the investigator which would likely lead to an additional project to be included in this dissertation.

8 Summary and Long-Term Vision

The long-term goal of our work is to produce tools, models and algorithms to enhance code understanding. In general, evaluation takes three forms: directly, by analyzing the model's accuracy with respect to training data; indirectly, by examining correspondence between the model's output and traditional quality metrics; and against humans, by comparing produced documentation to human-written documentation.

Correctly composing and maintaining trustworthy software involves understanding it at many levels and reliably answering a number of questions, including:

- How easy is it to understand and maintain this software? (code readability, Section 3.1.1)
- Where are the corner cases, and where are the common paths? (runtime behavior, Section 3.1.2)
- How can this code go wrong? (documenting exceptional situations, Section 3.2.1)
- How can I best use an off-the-shelf library? (documenting APIs, Section 3.2.2)
- What does a proposed fix really do? (documenting code changes, Section 3.2.3)

We claim that program understandability is a critical to successful software engineering. We propose to provide developers with a better understanding of programs.

References

[1] K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. *Reliability and Maintainability Symposium*, pages 235–241, Sept. 2002.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

[3] G. Alonso, C. Hagen, D. Agrawal, A. E. Abbadi, and C. Mohan. Enhancing the fault tolerance of workflow management systems. *IEEE Concurrency*, 8(3):74–81, July 2000.

[4] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Principles of Programming Languages*, 2005.

[5] S. Ambler. Java coding standards. *Softw. Dev.*, 5(8):67–71, 1997.

[6] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. *Workshop on Inspection in Software Engineering*, July 2001.

[7] G. Antoniol, M. D. Penta, H. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electr. Notes Theor. Comput. Sci.*, 127(3):87–99, 2005.

[8] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM Press.

[9] M. Arnold, S. J. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 52–64, 2000.

[10] T. Ball and J. R. Larus. Branch prediction for free. In *Programming language design and implementation*, pages 300–313, 1993.

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

[12] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *Principles of programming languages*, pages 134–148, 1998.

[13] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[14] R. Berg and D. Cruz. Security vulnerabilities in the spring framework model view controller. White paper, Ounce Labs, September 2008. Available online (21 pages).

[15] P. Berube and J. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Performance Analysis of Systems and Software*, pages 251–260, 2006.

[16] F. P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[17] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, 2006.

[18] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *International Conference on Software Engineering*, 2009.

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

[20] 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.

[21] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, 1997.

[22] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, H. Spencer, D. Keppel, , and M. Brader. *Recommended C Style and Coding Standards: Revision 6.0.* Specialized Systems Consultants, Inc., Seattle, Washington, June 1990.

[23] T. Cargill. Exception handling: a false sense of security. *C++ Report*, 6(9), 1994.

[24] 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.

[25] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[26] T. Copeland. *PMD Applied.* Centennial Books, Alexandria, VA, USA, 2005.

[27] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Trans. Comput. Syst.*, 26(4):1–68, 2008.

[28] M. Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation*, pages 35–46, 2000.

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

[30] 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.

[31] L. E. Deimel Jr. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, 1985.

[32] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, 1976.

[33] R. Dunn. *Software Defect Removal.* McGraw-Hill, 1984.

[34] 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.

[35] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[36] R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, 1948.

[37] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.

[38] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[39] R. Glass. *Facts and Fallacies of Software Engieering.* Addison-Wesley, 2003.

[40] J. B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[41] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Lecture Notes in Computer Science*, volume 2648, pages 121–135, Jan. 2003.

[42] N. S. R. W. Group. Software reuse survey. In *http://www.esdswg.com/softwarereuse/Resources/library/working_group_documents/survey2005*, 2005.

[43] R. Gunning. *The Technique of Clear Writing.* McGraw-Hill International Book Co, New York, 1952.

[44] R. Gupta, E. Mehofer, and Y. Zhang. Profile-guided compiler optimizations. In *The Compiler Design Handbook*, pages 143–174. 2002.

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

[46] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

[47] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[48] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.

[49] S. Huang and S. Tilley. Towards a documentation maturity model. In *International Conference on Documentation*, pages 93–99, 2003.

[50] R. Jain. Api-writing / api-documentation. In *http://api-writing.blogspot.com/*, Apr. 2008.

[51] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks*, pages 52–61, 2008.

[52] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–47, New York, NY, USA, 2005. ACM.

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

[54] M. Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance, ICSM*, pages 354–363, 2001.

[55] J. P. Kinciad and E. A. Smith. Derivation and validation of the automated readability index for use with technical materials. *Human Factors*, 12:457–464, 1970.

[56] J. C. Knight and E. A. Myers. Phased inspections and their implementation. *SIGSOFT Softw. Eng. Notes*, 16(3):29–35, 1991.

[57] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium*, pages 295–315, 2003.

[58] D. Krishnamurthy, J. A. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Softw. Eng.*, 32(11):868–882, 2006.

[59] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[60] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, June 9–11 2003.

[61] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.

[62] D. Malayeri and J. Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, pages 200–220, 2006.

[63] G. H. McLaughlin. Smog grading – a new readability. *Journal of Reading*, May 1969.

[64] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. Building more usable apis. *IEEE Softw.*, 15(3):78–86, 1998.

[65] P. Mehra and B. Wah. Synthetic workload generation for load-balancing experiments. *IEEE Parallel Distrib. Technol.*, 3(3):4–19, 1995.

[66] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[67] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, pages 85–103, 1997.

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

[69] A. Nanda and L. M. Ni. Benchmark workload generation and performance characterization of multiprocessors. In *Conference on Supercomputing*, pages 20–29. IEEE Computer Society Press, 1992.

[70] G. C. Necula and P. Lee. The design and implementation of a certifying compiler (with retrospective). In *Best of Programming Languages Design and Implementation*, pages 612–625, 1998.

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

[72] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*, pages 133–141, New York, NY, USA, 2002. ACM.

[73] N. I. of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, Research Triangle Institute, May 2002.

[74] D. Parnas. Software aging. In *Software Fundamentals*. Addison-Wesley, 2001.

[75] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2009. ACM.

[76] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

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

[78] D. R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16. IBM Press, 1991.

[79] P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. *Empirical Software Engineering, 2005. 2005 International Symposium on*, November 2005.

[80] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

[81] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Softw.*, 26(6):27–34, 2009.

[82] M. P. Robillard and G. C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Dept. of Computer Science, University of British Columbia, 1, 1999.

[83] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

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

[85] S. Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.

[86] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

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

[88] R. Selby. Enabling reuse-based software development of large-scale systems. *Software Engineering, IEEE Transactions on*, 31(6):495–510, June 2005.

[89] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[90] K. Sen. Concolic testing. In *Automated Software Engineering*, pages 571–572, 2007.

[91] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423, 2006.

[92] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Trans. Softw. Eng.*, 26(11):1101–1118, 2000.

[93] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[94] C. Simonyi. Hungarian notation. *MSDN Library*, November 1999.

[95] SpringSource. Spring framework, December 2008. `http://www.springframework.org/`.

[96] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: improving api documentation using usage information. In *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 4429–4434, New York, NY, USA, 2009. ACM.

[97] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison-Wesley Professional, 2004.

[98] B. Thomas and S. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *International Conference on Computer Documentation*, pages 235–236, 2001.

[99] A. Watters, G. van Rossum, and J. C. Ahlstrom. *Internet Programming with Python.* MIS Press/Henry Holt publishers, New York, 1996.

[100] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[101] W. Weimer and N. Mishra. Privately finding specifications. *IEEE Trans. Software Eng.*, 34(1):21–32, 2008.

[102] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, 2004.

[103] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.

[104] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.

[105] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, 2002.

[106] D. Zokaities. Writing understandable code. In *Software Development*, pages 48–49, jan 2002.

Ph.D. Proposal Addendum

Raymond P.L. Buse
buse@cs.virginia.edu

April 6, 2010

1 Introduction

We thank the committee for their thoughtful insights regarding this proposal. This addendum describes additional content to be included in the proposed dissertation as recommended by the committee. We address three areas: positioning, techniques, and evaluation. Positioning (Section 2) will address the committee's concerns with the motivation of the work and its relation to the proposed projects including the title of the work and the statement of thesis. In Section 3 we will discuss concerns related to proposed techniques; why they were chosen, the guarantees they can provide and how associated risks can be addressed. Finally, in Section 4 we elaborate on our proposed evaluation strategy; we address concerns of repeatability, comparison, and metrics.

2 Positioning

We propose to somewhat alter the positioning of the work, representing it as *Automatically Describing Program Structure and Behavior.* This title emphasizes the distinguishing characteristics of the work; the proposed techniques are fully automatic and can be readily employed to measure and characterize the way programs are written and how they behave. This leads us to a revised thesis statement:

> **Thesis:** *Program structure and behavior can be accurately modeled with semantically shallow features and human descriptions of these qualities can often be supplemented or replaced automatically.*

Our evaluation strategy includes testing two types of hypothesis: (1) if the models we propose accurately reflect program characteristics, and (2) if the output we generate typically contains the same information as corresponding human-written descriptions. For example, in *Readability* we test the hypothesis that a model of local code textual surface features correlates significantly (i.e., Spearman's ρ, $p - value < 0.05$) with human judgments of readability. In *Documentation for APIs* we test the hypothesis that machine generated exception documentation contains at least as much information (see Section 4.1) as existing JavaDoc documentation on average. We will include a secondary series of experiments to test for external validity: whether the features we model and output we generate correlates well with other notions of software quality.

Additionally, we will expand our current motivation concerning the importance of code reading and documentation to understanding programs and software engineering in general. We will discuss and cite experts like Knuth [8] and others [3, 13, 15]. who claim that reading code, in and of itself, is critically important and represents a large part of developer time and effort. We will expand references to experts like DeMarco [4] and studies like [5] which assert that documentation is a critical aspect of software quality. We will also add a number of citations to previous work in program understanding noted by the committee including work by Rossen [14] and Basili [16] as well as work on estimating program behavior for reasons including compiler optimization (eg., [1, 12, 20]).

3 Techniques

The proposed dissertation will elaborate on our choice of techniques as detailed in the associated publications but elided from the original proposal because of space limitations. Furthermore, we will include a cost/benefit analysis of each of our documentation algorithms, in particular as it relates to the presence and impact of false positives and false negatives. We discuss, for example, how each of our techniques can be adapted to ensure soundness (see Section 3.2).

3.1 Algorithm Selection

The proposed dissertation will elaborate on choice of technique in several dimensions. Here, we discuss our choice of modeling programs using semantically shallow features as opposed to heavyweight analyses. We also discuss our selection process for machine learning algorithms.

Our choice of semantically shallow features to model program understandability is a key insight of our work. This technique affords us multiple advantages over conventional heavyweight analyses. First, it allows us to simultaneously consider many disparate aspects of code, each of which contribute to judgments of behavior; conventional techniques depend on their ability to precisely characterize only a single program feature at a time. This idea is essential to the accuracy of our approach. Second, our lightweight analysis scales easily to even the largest software systems and can readily be adapted to many different languages; two features that prior approaches can rarely claim.

Our choice of classifiers to model our feature set is less critical. The proposed dissertation will describe the process by which we considered a large number of classifiers including neural networks, decision trees, and statistical techniques. We often found that many such classifiers were about equal in accuracy, compelling us to simply select the most efficient one.

3.2 Algorithm Soundness

The committee notes that our documentation tools have a potential to be misleading. The proposed dissertation will include discussions of how each tool can be tuned to produce output that guarantees soundness. Below we give examples of how we propose to deal with major sources of unsoundness in each documentation type. Other sources of error can be identified and addressed in a similar way.

Documentation for Exceptions relies on an underlying alias analysis to infer the precise set of Exceptions each method in a program can throw. Due to theoretical limitations, even the most precise alias analysis may lead our algorithm to output false positives (e.g., `foo() throws Exception e`, when in reality it cannot) and/or false negatives (e.g., `bar() throws nothing`, when in reality it may throw an exception). We propose to mitigate the danger of misleading developers by choosing a conservative (i.e., sound) alias analysis which will not produce any false negatives, but may produce false positives. Thus by using our tool, a developer can at worst be lead to introduce "dead code" in an attempt to handle an exception that will never happen. This strategy is similar to the practice of defensive programming.

Documentation for Program Changes includes a code summary component which can explicitly be lossy. This choice was made to mimic human documentation where not all code is considered equally important to document. Nevertheless, there are cases where developers may prefer a reliably sound output. We propose to provide this with a tool that allows users to either disable all lossy transformations and/or allow the algorithm to fall back on generic but sound output (e.g., `method foo() has changed`) when necessary.

Documentation for APIs can also be slightly modified to provide certain quantifiable guarantees. Our proposed algorithm infers a ranked list of possible usage documentations for a given class where each usage is presented with a confidence value derived from the frequency at which that pattern has been observed in the training corpus. While the proposed algorithm generally outputs the highest ranked example, we propose to provide a "mode" whereby the user can view the complete list of examples along with confidence values for each. Such a confidence value c guarantees that for all usages of the class in the training corpus, at least c of them match the usage described.

4 Evaluation

Finally, we address concerns voiced by the committee relating to the proposed evaluation. We discuss ensuring that our evaluation is repeatable and properly measured. First, we address our experiments involving comparison of machine generated documentation to human documentation. We propose two techniques to address threats to validity from bias and non-repeatability; an objective and repeatable comparison rubric, and the use of multiple annotators (Section 4.1). Second, we discuss metrics for inter-annotator agreement (Section 4.2).

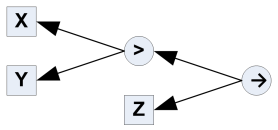

Figure 1: `if X>Y then Z` comprises a `greater-than` relation nested in an `implies` relation.

4.1 Documentation Comparison Rubric

We propose 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 machine-generated documentation, this step is automatic; the proposed algorithm outputs a list of objects. The human-written documentation is inspected by a human and each noun that refers to a specific program *variable*, *method*, or *class* is recorded. This criterion admits only *implementation* details, following the taxonomy of Sommerville [17], 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 are not considered program objects, and are not enumerated.

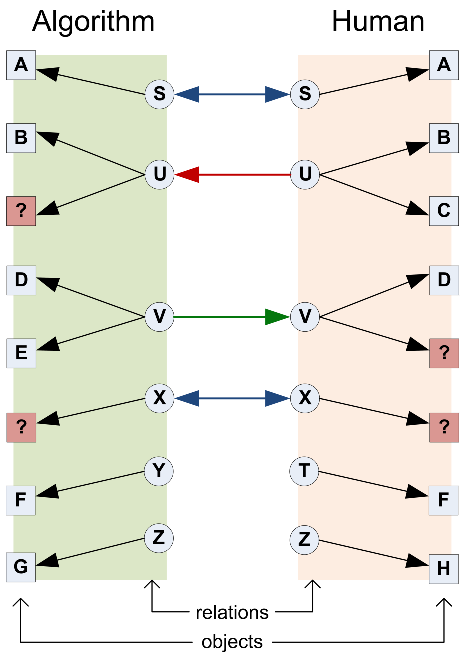

Figure 2: In documentation comparison, *objects* and *relations* are identified and mapped. A "?" indicates that the relation involves an unnamed object.

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 automatic for machine generated documentation. A human inspects 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 A.1 (see appendix), 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 sub-relations (e.g., Figure 1). The associated source is used to check the validity of each encoded relation. If a relation is discovered to be inconsistent with the artifact described, then the relation is removed from consideration. If the annotator cannot tell if the relation is consistent or not, then it is conservatively assumed to be consistent.

Step 3: For each relation in either the human- or algorithm-generated documentation, a human checks to see if the relationship is also fully or partially present in the other documentation. For each pair of relations

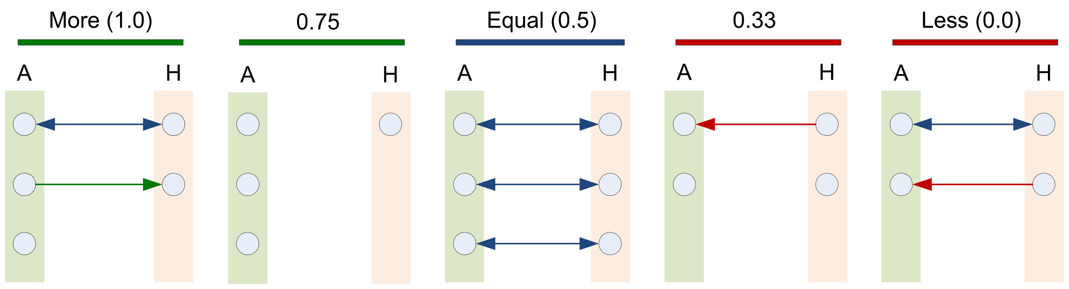

Figure 3: Example Algorithm (A) to Human (H) relation mappings.

H and A, the annotator determines if H *implies* A written $H \Rightarrow A$, or if $A \Rightarrow H$. As shown as a set of rule templates in Figure 2, a relation *implies* another relation if the relation operator is the same and each operand corresponds to either the same object or an object that is implicit and unnamed (see below for an example of an unnamed object). If the relations are identical, then $H \Leftrightarrow A$. Again, the annotator is given access to 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 conference 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 – see below). Automated approaches to this problem have been proposed (e.g., [11, 18]), however we use human annotators who have been shown to be accurate at this task [2].

After annotation is complete, we can quantify the information relationship between two documentations as in Figure 3. If the mapping is a bijection, then documentations are considered to contain *equal* information (0.5). If the mapping is found to be a surjection from A to H, (i.e., all single headed arrows are right arrows, and all human relations are mapped) then the algorithm documentation is considered to contain strictly *more* information (1.0). If the mapping is surjective from H to A (i.e., all single headed arrows are left arrows, and all algorithm relations are mapped) then the algorithm documentation is considered to contain strictly *less* information (0.0). In other cases, we throw out all relations that are directly subsumed by another relation. We then divide the number of relations in the algorithm documentation by the total number of relations in both. Conceptually, this expresses the fraction of all the information documented that was documented by the algorithm.

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. Furthermore, this process allows us to precisely quantify the difference in information content. This technique is similar to the NIST ROUGE metric [10] for evaluating machine-generated document summaries against human-written ones. In addition to the direct results, we will quantify how often a group of independent annotators agreed on the assessment presented.

For example:

```
Human:  has an insufficient amount of gold
Algorithm:  getPriceForBuilding() > getOwner().getGold()
```

To compare these documentations, we first enumerate object sets. For the human documentation this consists of {`gold`}, and for the algorithm {`building price, gold`}. The relationships consist of: for the human documentation {`gold` < `?`} because "insufficient" implicitly defines a less-than relation, and for the algorithm documentation {`gold` < `building price`}. Finally, we find that the algorithm documentation *implies* the human documentation because if `gold` < `building price` then it follows that `gold` < something.

4.2 Inter-annotator agreement

We also address the evaluation metric for inter-annotator reliability, particularly on a 5-point "Likert" scale [9]. Previously, we have used Pearson's r, Spearman's ρ, and Cohen's κ (and associated significance tests) to measure the agreement between two annotators. To measure agreement between a group of annotators we compute all pair-wise agreements and average them [7, 19]. This technique is similar to other previous work (e.g., [6, 21]). The proposed dissertation will include a discussion of other possible metrics as well as statistical techniques involving repeated measures with likert style scales.

5 Conclusion

The proposed work will include in-depth discussions of other threats to validity not addressed in the original proposal for space reasons. While we have endeavored to narrow or otherwise codify a number of claims from the original proposal, we will continue to revise and expand on the connection between the proposed work and software engineering tasks both by citing experts in the field and by conducting our own investigations. We again thank the committee for their suggestions for improving the proposed work and its presentation.

A Table of Relational Operators

Type	Arity	Symbol	Name
logical	unary	T	is *true* / *always*
		F	is *false* / *never*
	binary	$\vee$	or
		$\wedge$	and
		$=$	equal to
		$\neq$	not equal to
		$<$	less-than
		$>$	greater-than
		$\leq$	less-than or equal to
		$\geq$	greater-than or equal to
programmatic	unary	$\emptyset$	is empty
		$()$	call / invoke
		$\circlearrowright$	`return`
		$\uparrow$	`throw`
	binary	$\leftarrow$	assign to
		$\in$	element of
		$\Rightarrow$	implies
		$:$	instance of
edit	unary	$+$	added
		Δ	changed
		$-$	removed

Table A.1: Table of relational operators used for documentation comparison. Together these form the basis of our documentation description language. *Logical* operators are used to describe runtime conditions, *programmatic* operators express language concepts, and *edit* operators describe changes to high-level structure (classes, fields, and methods).

References

[1] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Softw. Eng. Notes*, 29(4):195–205, 2004.

[2] C. Cardie and K. Wagstaff. Noun phrase coreference as clustering, 1999.

[3] L. E. Deimel Jr. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, 1985.

[4] T. DeMarco. *Why Does Software Cost So Much?* Dorset House, 1995.

[5] M. Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 354, Washington, DC, USA, 2001. IEEE Computer Society.

[6] J. Kashden and M. D. Franzen. An interrater reliability study of the luria-nebraska neuropsychological battery form-ii quantitative scoring system. *Archives of Clinical Neuropsychology*, 11:155–163, 1996.

[7] M. G. Kendall and B. B. Smith. The problem of m rankings. *The Annals of Mathematical Statistics*, 10(3):275287, 1939.

[8] D. Knuth. Literate programming (1984). In *CSLI*, page 99, 1992.

[9] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:44–53, 1932.

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

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

[12] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.

[13] D. R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16. IBM Press, 1991.

[14] M. B. Rosson. Human factors in programming and software development. *ACM Comput. Surv.*, 28(1):193–195, 1996.

[15] S. Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.

[16] F. Shull, I. Rus, and V. Basili. Improving software inspections by using reading techniques. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 726–727, Washington, DC, USA, 2001. IEEE Computer Society.

[17] I. Sommerville. *Software Engineering (7th Edition).* Pearson Addison Wesley, 2004.

[18] 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.

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

[20] D. W. Wall. Predicting program behavior using real or estimated profiles. *SIGPLAN Not.*, 26(6):59–70, 1991.

[21] Y. S. Wasfi, C. S. Rose, J. R. Murphy, L. J. Silveira, J. C. Grutters, Y. Inoue, M. A. Judson, and L. A. Maier. A new tool to assess sarcoidosis severity. *CHEST*, 129(5):1234–1245, 2006.