Abridging Source Code

BINHANG YUAN, Rice University, U.S.A
VIJAYARAGHAVAN MURALI, Rice University, U.S.A
CHRISTOPHER JERMAINE, Rice University, U.S.A

In this paper, we consider the problem of source code abridgment, where the goal is to remove statements from a source code in order to display the source code in a small space, while at the same time leaving the "important" parts of the source code intact, so that an engineer can read the code and quickly understand purpose of the code. To this end, we develop an algorithm that looks at a number of example, human-created source code abridgments, and learns how to remove lines from the code in order to mimic the human abridger. The learning algorithm takes into account syntactic features of the code, as well as semantic features such as control flow and data dependencies. Through a comprehensive user study, we show that the abridgments that our system produces can decrease the time that a user must look at code in order to understand its functionality, as well as increase the accuracy of the assessment, while displaying the code in a greatly reduced area.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools*;

Additional Key Words and Phrases: Program Comprehension, Source Code Abridgment.

ACM Reference Format:
Binhang Yuan, Vijayaraghavan Murali, and Christopher Jermaine. 2017. Abridging Source Code. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 58 (October 2017), 26 pages. https://doi.org/10.1145/3133882

1 INTRODUCTION

In this paper, we consider the problem of source code abridgment, where the goal is to remove statements from a source code in order to display the source code in a small space, without compromising the ability of a reader of the code to understand, at a high level, the purpose of the source code. For some applications, the required abridgment will be radical, resulting in an abridged code that is 50%, 10% or only 5% of the size of the original code.

For an example of code abridgment, consider the Java method for merging two lists of integers into a single list, given in Listing 1. An abridged version of this code is given in Listing 2. Notice that the abridged code is no longer correct in the sense that the code would no longer correctly merge two lists, and (after removing the ellipses) it may not even compile. Still, in some sense, the abridged code communicates the "essence" of the algorithm that has been implemented: there is a loop that checks which of two values in an array is smaller (`temp[i] <= temp[j]`) and then, depending upon the result, the smaller value is copied into `array[k]`. Ideally, a human developer

Authors' addresses: `{by8, vijay, cmj4}@rice.edu`, Department of Computer Science, Rice University, 6100 Main Street, Houston, TX, 700005, USA.
Authors' addresses: Binhang Yuan, Computer Science Department, Rice University, 6100 Main Street, Houston, Texas, 77251, U.S.A, by8@rice.edu; Vijayaraghavan Murali, Computer Science Department, Rice University, 6100 Main Street, Houston, Texas, 77251, U.S.A, vijay@rice.edu; Christopher Jermaine, Computer Science Department, Rice University, 6100 Main Street, Houston, Texas, 77251, U.S.A, cmj4@rice.edu.

2475-1421/2017/10-ART58
https://doi.org/10.1145/3133882

looking at the abridged code should be able to understand the basic idea of what is going on just as easily—or even more easily—than had s/he browsed the original code. Further, it is possible to display the abridged code in a smaller physical space on a screen.

There are many applications of source code abridgment. Our particular motivation for studying this problem is that we believe that source code abridgment is a vital component of any source code search system. Just as Internet search results displayed by an Internet search engine such as Google pick out the important content from a web page to display to the consumer, we believe that the basic interface to a code search engine needs to do the same: it should display the methods/functions/classes most relevant to the search in a condensed fashion, so that a user can scroll through the search results without being overwhelmed by a huge number of codes.

```
private void merge(int low, int mid, int high){
      for (int i = low; i <= high; i++) {
            temp[i] = array[i];
      }
      int i = low;
      int j = mid + 1;
      int k = low;
      while (i <= mid && j <= high) {
            if (temp[i] <= temp[j]) {
                  array[k] = temp[i];
                  i ++;
            } else {
                  array[k] = temp[j];
                  j ++;
            }
            k ++;
      }
      while (i <= mid) {
            array[k] = temp[i];
            k ++;
            i ++;
      }
}
```

Listing 1. Original merge code.

```
private void merge(int low, int mid, int high){ ...
      while (i <= mid && j <= high) {
            if (temp[i] <= temp[j]) {
                  array[k] = temp[i]; ...
            } else {
                  array[k] = temp[j];  ...
            } ...
      } ...
}
```

Listing 2. Abridged merge code.

There have been a few research efforts in the software engineering community related to source code abridgment. *Program slicing* [Weiser 1981] allows a developer to extract a subset of the program, referred to as a *program slice*, that is relevant to a statement and a group of variables, called the *slicing criterion*. The slice is an executable program where the slicing criterion performs the same computation as the original program. However, this technique is mainly applied to code inspection and debugging, and relies on a knowledgeable developer being able to specify the slicing criterion. Asking an engineer not familiar a set of code search results to provide appropriate slicing criterion seems unreasonable.

Another popular approach to aid developers comprehend programs is to automatically generate a natural language summary for the source code [Abid et al. 2015; Eddy et al. 2013; Haiduc et al. 2010a,b; Liu et al. 2014; McBurney et al. 2014; McBurney and McMillan 2014, 2016; Moreno 2014; Moreno et al. 2013; Sridhara et al. 2010, 2011a]. For example, Sridhara et al. [2010] describe a technique to generate Java method documentation by first selecting some statements from the method body, then extracting keywords from the identifier names, and finally applying a natural language generator to stitch the keywords into English sentences, which will be used to generate the method summary. While this is a useful approach, and an English summarization may be a key component of the "ultimate" source code search engine, we believe that automatically describing code in English is not a substitute for actually displaying the code. After all, the developer is searching for code, and not an English document, and at some point the developer is going to want to see actual code.

Perhaps the effort most closely related to source code abridgment is the work on automatic source code folding [Fowkes et al. 2014]. This approach applies topic modeling to automatically determine whether to display or hide a block in the Java source code. However, source code folding fundamentally differs from code abridgment in that code folding automatically chooses to hide or display each block of code (for example, a set of statements within curly braces in Java). Source code folding is a much more constrained problem than source code abridgment, since an abridger can choose to remove any statement in the code. Considering, for example, the preceding `merge` example, block-level-only code hiding is a significant limitation if the goal is a fairly radical abridgment of the code. In this example, it is impossible to remove more than eight lines of code via folding without entirely removing the contents of the central `while` loop—and doing this would seem to obfuscate the purpose of the code. Further, via folding, it is impossible to reduce `merge` to less than ten lines of code without removing the entire method body.

Our Contributions. Specific contributions of the paper are:

- We present an algorithm that treats the problem of abridging a particular code as a constrained optimization problem. Given a source code, for each statement $i \leq n$ where n is the number of statements in the code, a vector of features x_i is extracted that describes the ith statement. Further, a vector of feature weights ω is learned that weights the features according to how strongly they indicate that a statement should be retained. Then the problem of abridging the code is reduced to choosing a set of statements that maximize the sum of $x_i^T \omega$ over all retained statements subject to a set of constraints derived from the syntax and semantics of the code (T is the vector transpose operation here, and $x_i^T \omega$ is the inner product of the feature vector x_i and the coefficient vector ω).
- We frame the problem of learning the weight vector ω as an optimization problem, where the goal is to maximize the performance of the resulting algorithm on a set of training codes that have been hand-abridged by experts.
- We perform a comprehensive user study which gives strong evidence that automatically abridged code can actually facilitate a greater level of understanding in less time than the original code, in addition to the obvious benefit that the code is shorter.

2 OVERVIEW

At the highest level, the source code abridgment procedure we propose retains a subset of statements from a program in order to help users understand the overall functionality of the code. We encode this code abridgment problem as a constrained optimization procedure that can be solved by an integer programming (IP) solver, which attempts to maximize the value of an objective function $f(\Delta|\omega, P)$, where Δ encodes the subset of statements from program P retained by the abridgment, and ω parameterizes the function. f measures the quality of the abridgment, attempting to answer a series of questions regarding the quality of the abridgment (Does the abridgment select the most informative statements? Does it keep variables that are used often? Are statements with important identifiers chosen for inclusion in the program?). These questions are answered by encoding key aspects of each program statement as a feature vector (aspects of a statement that are encoded in the feature vector include the type of statement, the parent statement type, the level at which the statement is nested in the program, and so on; features will be described in detail in Section 4 of the paper). The inner product of the feature vector and the weight vector ω produces a weighted sum of the various features. This weighted sum then measures how important it is to retain the statement in the abridged program. At the same time, an abridgment produced must follow certain constraints derived from the syntax and semantics of the original code.

Crucially, the goodness-of-fit function will rely on the inner product between the various feature vectors and a number of weights—contained in the vector ω—that are difficult to choose manually. For example, how can we choose a weight that informs the IP solver how important is it for a `for` loop to remain in a program? Is that weight larger or smaller than the weight governing the inclusion of a `return` statement in the abridged code? Since it is unclear how a system designer can make such choices, we instead choose the weights automatically, by learning from data.

Thus, choosing the weight vector ω is really an algorithmic parameter selection problem, as we are choosing a set of parameters that make the underlying abridgment algorithm perform optimally. This requires an "optimization-within-optimization" procedure, where we need to determine the best way to formulate an optimization procedure (the IP, where we must choose the weight vector ω), via an outer optimization procedure. Such problems can be difficult, as there is no way to determine the performance of the algorithm given a set of input parameters without running it—this is in contrast to standard machine learning applications where it is typically possible to use partial derivatives to power a steepest descent solver. Further, various combinations of parameters can have unexpected results on algorithm performance. Fortunately, high-quality solvers for such problems do exist. These solvers generally work using a local search algorithm that carefully selects sets of training examples on which to run the underlying algorithm, in order to determine the effect of various parameter configurations. In our implementation, we rely on ParamILS [Hutter et al. 2009, 2007].

To use such a solver to power an optimization-within-optimization procedure, we need to be able to measure the quality of the parameterized abridgment algorithm. To measure abridgment quality, we adopt a supervised approach. We first choose a number of example codes from a diverse set of sources: JSON parsers, natural language processing libraries, database implementations, web crawlers, and so on, to produce a training set. We then ask a human abridger to manually reduce the various codes in a way that the abridged codes retain enough information that another programmer can also easily understand the high-level functionality of this Java method. Given this set of training examples, the goal is to use the optimization-within-optimization procedure to produce an abridgment algorithm that most closely mimics the human-produced abridgment.

Choosing the weights so that the automatically-produced abridgments match the training set relies on having a way to measure the difference between two abridgments. Many options for measuring such a distance exist; we use the Jaccard distance. Let Δ_k be the set of statements retained by the automated abridgment procedure for training program k, and Δ_k^h be the set of statements retained by a human being on the same program, where the automated procedure has been constrained to retain statements with the same number of lines as the human being chose.

Then the Jaccard distance between these two sets can be computed as:

$$D_{Jac}\left(\Delta_k, \Delta_k^h\right) = \frac{\left|\Delta_k \cup \Delta_k^h\right| - \left|\Delta_k \cap \Delta_k^h\right|}{\left|\Delta_k \cup \Delta_k^h\right|} \tag{1}$$

and the weight vector ω is chosen so as to minimize $L(\omega)$, the average of the Jaccard distances over all M training programs:

$$L(\omega) = \frac{1}{M}\sum_{k=1}^{M} D_{Jac}\left(\arg\max_{\Delta_k} f(\Delta_k|\omega, P_k), \Delta_k^h\right) \tag{2}$$

Since it is expensive to produce human truncated codes for training, we find that over-fitting the training set provided to the optimization-within-optimization is a significant problem. For a simple example of the pitfalls of over-fitting while learning an abridger, relatively few codes in our expert-provided training abridgments contain the Java `LabelStatement` construct, as relatively few labels are used in modern programming. If all such constructs are removed in the example abridgements, the learner is free to choose an extreme weight that makes it impossible for such a statement to ever be included in an abridged program. A much better tactic would be to choose a weight that is just small enough to ensure that all `LabelStatement` constructs are removed from the training programs, rather than choosing a weight so extreme that no such construct can ever appear. This can be accomplished via the addition of a L_2 *regularization* term to the loss function, so the final form of our loss function is:

$$L(\omega) = \frac{1}{M}\sum_{k=1}^{M} D_{Jac}\left(\arg\max_{\Delta_k} f(\Delta_k|\omega, P_k), \Delta_k^h\right) + \lambda \left\|\omega\right\|_2^2 \tag{3}$$

where λ controls the extent of the regularization. Intuitively, the optimization-within-optimization is now penalized for choosing weights of large magnitude.

When training our abridger, we use a training set of 70 Java methods. These methods have been chosen to present a diverse set of codes to learn from. In particular, we chose methods with a high variance in original (unabridged) lengths, and when preparing the training set we abridged each training program at a variety of reduction rates. When using ParamILS to tune ω, we run $100,000$ iterations of ParamILS local search over the training set. This takes around 6 hours using a 1.1 GHz dual core CPU and 8 GB Memory. Since training is an offline process, the relatively long training time, even for this small training set, is not much of a concern.

3 IP FORMULATION

In this section, we define the code abridgment problem.

Our current work focuses on abridging Java methods. Our code abridgment system views a Java method as a graph in which the nodes are statements and an edge from statement i to statement j represents a constraint such that if statement j appears in the abridgment, then statement i must (or possibly, should) appear in the abridgment as well.

We apply a standard Java parser (the Eclipse AST framework) to extract the statements from the Java method to abridge. A statement can either be a simple statement without children, e.g., an `ExpressionStatement`, or a more complicated statement with children, e.g., a `WhileStatement`. For statements with children, we only include the non-recursive part of the statement in the node. For example, for a `WhileStatement`, only the control condition is retained, while the statements nested inside the body block are represented as independent nodes in the graph.

When we analyze a program to perform an abridgment, we associate two values with statement i in the graph: a vector of features x_i, and a line count l_i. As described in the previous section, x_i is designed so that the inner product of x_i and a vector of feature weights ω measures the relative importance of retaining this statement in the abridged program. l_i represents a canonical count of how many lines are required to display the ith statement in the code. This is chosen so as to match the canonical number of lines typically used to display a statement. For example, in the `merge` method in Listing 1, the line count for the `ForStatement` in line 2 is two, because the `ForInitializationExpression`, `ConditionExpression` and `ForUpdateExpression` take one line in the canonical listing, and the back bracket in line 3 takes one line. Note that the `ExpressionStatement` inside the body of this `ForStatement` does not contribute to the line count of the `ForStatement`.

In our abridgment framework, we define two types of graph edges: those representing *hard constraints*, E_H, and those representing *soft constraints*, E_S. An edge from i to j corresponding to a hard constraint encodes a type of dependence between the two statements that implies that if statement j is present in the abridgment, statement i must be retained as well. An edge from i to j corresponding to a soft constraint encodes the dependence between two statements that should be retained in the abridgment, if possible. If the soft constraint is violated and statement j is included but statement i is not, we will add a penalty and the abridgment will not be scored as high.

Finally, an abridgment task is given a line count range $[L_{low}, L_{high}]$; the total sum of l_i values corresponding to statements retained by the abridgment must fall within this range.

Then, given a program P represented as a graph of statements (each of which has an associated l_i and x_i value) with edge sets E_S and E_H, the optimal abridgment of a program is a vector of binary variables $\Delta = \langle \delta_1, \delta_2, ..., \delta_n \rangle$ with one δ_i value per statement. This vector is chosen so as to solve the integer program

$$\arg\max_{\Delta} f(\Delta|\omega, P)$$

where $f(\Delta|\omega, P) =$

$$\left(\sum_i \left(x_i^T \omega\right) \cdot \delta_i - \alpha \sum_{e(i,j)\in E_S} \left(\delta_j - \delta_i\right)\right) \text{ if } \left\{ \begin{array}{l} \forall e\,(i,j) \in E_H \wedge \delta_j = 1 \Rightarrow \delta_i = 1 \\ L_{low} \le \sum_i l_i \delta_i \le L_{high} \end{array} \right. \tag{4}$$

and $f(\Delta|\omega, P) = -\infty$ otherwise.

4 CONSTRAINTS AND FEATURE ENGINEERING

Thus far, we have described our basic optimization-within-optimization procedure, as well as how the optimization problem corresponding to the abridgment is phrased as an IP problem. However, we have been a bit coy as to the details. Specifically: what are the hard and soft constraints that we actually use in our implementation? And how do we analyze a program statement and extract a feature vector from it? In this section, we address both of those questions.

4.1 Example

Before tackling the question of constraint and feature engineering, we first give an example of the sort of abridgments that our system is able to produce using the constraints and features described in this section.

Consider the Java method performing an SMTP protocol connection attempt that is is shown in Listing 3. This method is passed a user name and password and then attempts authentication using these values.

```java
private boolean authenticate(String user,String passwd) throws MessagingException {
        String mechs=session.getProperty("mail." + name + ".auth.mechanisms");
        if(mechs == null)
                mechs=defaultAuthenticationMechanisms;
        String authzid=getAuthorizationId();
        if(authzid == null)
                authzid=user;
        if(enableSASL){
                if(debug)
                        out.println("DEBUG SMTP: Authenticate with SASL");
                if(sasllogin(getSASLMechanisms(),getSASLRealm(),authzid,user,passwd))
                        return true;
                if(debug)
                        out.println("DEBUG SMTP: SASL authentication failed");
        }
        if(debug){
                out.println("DEBUG SMTP: Attempt to authenticate");
                out.println("DEBUG SMTP: check mechanisms: " + mechs);
        }
        StringTokenizer st=new StringTokenizer(mechs);
        while(st.hasMoreTokens()){
                String m=st.nextToken();
                String dprop="mail." + name + ".auth."+ m.toLowerCase(Locale.ENGLISH)+ ".disable";
                boolean disabled=PropUtil.getBooleanSessionProperty(session,dprop,false);
                if(disabled){
                        if(debug)
                                out.println("DEBUG SMTP: mechanism " + m + " disabled by property: "+ dprop);
                        continue;
                }
                m=m.toUpperCase(Locale.ENGLISH);
                if(!supportsAuthentication(m)){
                        if(debug)
                                out.println("DEBUG SMTP: mechanism " + m + " not supported by server");
                        continue;
                }
                Authenticator a=(Authenticator)authenticators.get(m);
                if(a == null){
                        if(debug)
                                out.println("DEBUG SMTP: " + "no authenticator for mechanism " + m);
                        continue;
                }
                return a.authenticate(host,authzid,user,passwd);
        }
        throw new AuthenticationFailedException("No authentication mechanism supported by server & client");
}
```

Listing 3. A Java method performs the SMTP protocol connection attempt.

We show two abridgments of this code. In the first abridgment, given as Listing 4, we have used our tool to reduce the size to approximately 50% of the number of lines in the original code. Interestingly, in this first abridgment, all of the debugging statements have been removed from the code. Debugging statements are typically the first type of statement removed by a human abridger from a training code, and the automatic abridger has learned to mimic this.

```java
private boolean authenticate(String user,String passwd) throws MessagingException {
        String mechs=session.getProperty("mail." + name + ".auth.mechanisms");...
        String authzid=getAuthorizationId();
        if(authzid == null)
                authzid=user;
        if(enableSASL){...
                if(sasllogin(getSASLMechanisms(),getSASLRealm(),authzid,user,passwd))
                        return true;...
        }...
        StringTokenizer st=new StringTokenizer(mechs);
        while(st.hasMoreTokens()){
                String m=st.nextToken();
                String dprop="mail." + name + ".auth."+ m.toLowerCase(Locale.ENGLISH)+ ".disable";
                boolean disabled=PropUtil.getBooleanSessionProperty(session,dprop,false);
                if(disabled){...
                        continue;
                }
                m=m.toUpperCase(Locale.ENGLISH);
                if(!supportsAuthentication(m)){...
                        continue;
                }
                Authenticator a=(Authenticator)authenticators.get(m);...
                return a.authenticate(host,authzid,user,passwd);
        }...
}
```

Listing 4. Abridge the `authenticate` Java method to around 50% of the original length.

In the second abridgment, the length of the code has been further reduced to be only 20% of the length of the original code. It is interesting that the abridger has cut the code down to only its essential elements. The abridgment spends four lines showing the attempt to obtain the authorization mechanisms and identifier, and then the check of whether SASL is enabled. It then immediately shows the method's central loop, which parses the authorization mechanism string, processing one token at a time. Some important information has been dropped (such as the fact that the string being tokenized comes from the authorization mechanisms that have been obtained) but this is still a reasonable choice, as the lines saved allow the abridger to show the important details of how the authentication is performed.

```java
private boolean authenticate(String user,String passwd) throws MessagingException {
        String mechs=session.getProperty("mail." + name + ".auth.mechanisms");...
        String authzid=getAuthorizationId();...
        if(enableSASL){...
        }...
        while(st.hasMoreTokens()){
                String m=st.nextToken();
                String dprop="mail." + name + ".auth."+ m.toLowerCase(Locale.ENGLISH)+ ".disable";
                boolean disabled=PropUtil.getBooleanSessionProperty(session,dprop,false);...
                Authenticator a=(Authenticator)authenticators.get(m);...
                return a.authenticate(host,authzid,user,passwd);
        }...
}
```

Listing 5. Abridge the `authenticate` Java method to around 20% of the original length.

4.2 Constraints

We now describe the hard and soft constraints used to obtain this abridgment.

Hard Constraints. Both the hard and soft constraints used by our abridgment system make use of a program dependence graph [Ferrante et al. 1987] extracted from the method to be truncated. A typical program dependence graph contains both the data and control flow dependences for each operation in the method; we choose to encode control flow dependencies as hard constraints. Intuitively, the reason for this is that if control flow dependencies are not respected in the final abridgment presented to the user, the resulting abridgment may not be simply incomplete (which is unavoidable when radically abridging a code), but it may actually be *misleading* to a reader of the code, which we would like to avoid. For example, it makes little sense to remove an `IfStatement` from a program while retaining the action to be taken if the `IfStatement`'s prediction evaluates to true.

For a real life example of the sort of problems that can result from not respecting such dependencies, consider the abridgment that our system produced *without* using control flow dependencies as hard constraints, which is given in Listing 7. This is an abridgment of the original code in Listing 6. Without encoding control dependences between the `expression` and `break` statements in lines 8-9 and the `switch case` in line 7, the resulting abridgment appears to associate the move corresponding to `Paper` with `case 0`, which is misleading. In fact, `Paper` is associated with `case 1`. Encoding the control flow dependency graph prevents this sort of misleading abridgment.

```
 1 private String swithExample(){
 2         String computerMove;
 3         switch ((int)(3*Math.random())) {
 4                 case 0:
 5                         computerMove = "Rock";
 6                         break;
 7                 case 1:
 8                         computerMove = "Paper";
 9                         break;
10                 default:
11                         computerMove = "Scissors";
12                         break;
13         }
14         return computerMove;
15 }
```

Listing 6. A simple method that includes a `SwitchStatement`.

```
 1 private String swithExample(){
 2         String computerMove;
 3         switch ((int)(3*Math.random())) {
 4                 case 0:...
 5                         computerMove = "Paper";
 6                         break;
 7                 default:
 8                         computerMove = "Scissors";
 9                         break;
10         }
11         return computerMove;
12 }
```

Listing 7. A misleading abridgment for the simple method that includes a `SwitchStatement`.

Soft Constraints. We initially considered encoding data dependencies, such as loop-carried and loop-independent data dependences, as hard constraints. However, after scrutinizing the hand-truncated code in the training set, it appeared to us that while data dependencies are sometimes respected by a human abridger, they are still violated periodically. For example, it is not unusual for a human abridger to remove the `ExpressionStatement` that increases the value of counter variable inside a loop, which would not be allowable were data dependencies encoded as hard constraints. Consider the example code shown in Listing 1. The `ExpressionStatements` inside the first `while` loop are removed from lines 11, 14, and 16 in the abridgment in Listing 2. This violates the data dependence in the `WhileStatement` in line 8 on variables `i` and `j`, the dependence in the

IfStatement in line 9 on variables i and j, and the dependence in the ExpressionStatements in line 10 and 13 on variables k, i, j. However, removing these ExpressionStatements does not seriously compromise understandability.

We considered encoding all such data dependencies as soft constraints, but after some experimentation, we found that even this is not necessary. However, we did find that in expert-produced abridgments, when a variable declaration (usually with a complicated initializer) is removed, then all (or at least, most) statements referencing that variable are removed as well. This is not always the case; in the SMTP example at the beginning of this section, the declaration of the loop control variable st has been removed from the code, which is arguably the right decision. Hence we eventually settled on including all variable declaration dependencies as soft constraints. That is, there is an edge in the soft constraint graph from a variable declaration statement to each of the statements that utilize that variable and the abridger is penalized for removing a variable declaration while retaining statements using the variable.

4.3 Features

Clearly, constraints alone are not enough to produce a high-quality abridgment. The abridger must also make "softer" judgments regarding which statements are more important to keep in the abridged code. This is accomplished in our automatic abridger by taking the inner product $x_i^T \omega$ of the feature vector describing the ith statement x_i with the weight vector ω. In this subsection, we describe the feature engineering used to extract a feature vector from a statement.

Basic Procedure. Before describing each of the features actually used in our abridgment system, we begin by describing the process used to engineer features. In general, our tactic was to examine the abridgments produced by the system with a given feature set. By examining the codes, we would attempt to determine precisely what were the failures in the abridgments produced, and then develop a new feature that seemed to address the shortcoming in the current set of abridgments. For example: if it appears that debug statements—which are seemingly unimportant—are always being retained, it might make sense to add a feature that takes into account the actual name of all variables occurring in a statement. A variable named debug may indicate a debugging statement that can be removed.

For each new feature added, we would re-train the model and then examine the total Jaccard distance between the abridgments produced by running the abridger on the training set, and the actual abridgments in the training set. This provides an objective measure of the utility of an additional feature. If the distance goes down after adding the feature, it implies that the abridger is able to use the feature to produce abridgments that better mimic those present in the training set. Using such a measure of feature quality, as long as the distance decreases, there is relatively little risk of adding poor features; most of the risk along those lines would be associated with adding features that allow the abridger to over-fit the training data, or that are useful for learning to abridge only the training data, and do not work as well over a diverse set of real-life application cases.

For each of the features listed in this subsection, the total Jaccard distance does indeed go down when the feature is added, which is indicative of the utility of each of the features that we chose. As our training corpus grows, or if additional failure modes for the abridgment tool are discovered, additional features can be added and evaluated in the same way.

We now describe the set of features used by our tool.

Statement Type. In our initial development, we first attempted to only encode the type of the statement in the vector x_i. Each of the 22 different statement types from the Eclipse AST were mapped to a different binary feature (AssertStatement, Block, BreakStatement, ConstructorInvocation, `ContinueStatement`, etc.). This makes it possible to learn an abridgment procedure capturing the basic intuition, that, for example, control flow statements (`IfStatement`, `WhileStatement`) are typically more important to maintain in an abridgment than are error handling statement such as `AssertStatement` and `ThrowStatement`.

Not surprisingly, only encoding such feature turns out to be far from adequate. Consider the `handleDeclaration` method below.

```
 1 public CssProperty handleDeclaration(String property, CssExpression expression, boolean important) throws Exception {
 2       if(expression==null){
 3             throw new Exception("Null expression!");
 4       }
 5       CssProperty prop;
 6       if(Util.onDebug){
 7             System.err.println("Creating " + property + ": "+ expression);
 8       }
 9       try {
10             if(getMediaDeclaration().equals("on") && (getAtRule() instanceof AtRuleMedia)){
11                   prop=properties.createMediaFeature(ac,getAtRule(),property,expression);
12             } else {
13                   prop=properties.createProperty(ac,getAtRule(),property,expression);
14             }
15       } catch (InvalidParamException e){
16             throw e;
17       } catch (Exception e){
18             if(Util.onDebug){
19                   e.printStackTrace();
20             }
21             throw new InvalidParamException(e.toString(),ac);
22       }
23       if(important){
24             prop.setImportant();
25       }
26       prop.setOrigin(origin);
27       prop.setInfo(ac.getFrame().getLine(),ac.getFrame().getSourceFile());
28       return prop;
29 }
```

Listing 8. The `handleDeclaration` method creates a new property and assigns the expression to it.

Using only the statement type to abridge this code to approximately 50% of the original number of lines results in abridged version shown in Listing 9.

The resulting abridged code is quite poor. The abridgment procedure (arguably incorrectly) has chosen to remove the most important portion of the code (original lines 10 to 14) while retaining uninformative debugging and exception handling code, and leaving the overall (and relatively uninformative) control-flow skeleton of the code intact (`if`s, `try`s, `return`s, and `catch`es). The reason for this undesirable result is that our learning process recognizes that `IfStatement`, `TryStatement` and `ReturnStatement` should have relatively high weights—which is reasonable since more often than not such statements contain important information—and so these statements are retained while the body of each block is simply removed, without regard to its contents. This is problematic, because the reason that such statements should be retained is because of the important code contained *within* such statements. Blindly keeping each `IfStatement` and removing its contents results in an uninformative abridgment.

```
public CssProperty handleDeclaration(String property, CssExpression expression, boolean
      important) throws Exception {
       if(expression==null){...
       }
       CssProperty prop;
       if(Util.onDebug){...
       }
       try {...
       } catch (InvalidParamException e){
       } catch (Exception e){
             if(Util.onDebug){...
             }...
       }
       if(important){...
       }...
       return prop;
}
```

Listing 9. A poor abridgment of the handleDeclaration method.

Parent Statement Type. To address the deficiency illustrated in the previous example, we decided to include not just the statement type, but also the parent statement type as a feature, which provides important context for each statement. For example, statements nested inside the CatchClause of a TryStatement are generally designed only to handle a runtime exception, and do not contribute significantly to human understanding of a program. On the other hand, statements nested inside the body block of a TryStatement usually implement the main logic of a Java method.

To realize the parent statement type feature, we define thirteen parent statement types produced by the Eclipse Java parser: MethodDeclaration, DoStatement, EnhancedForStatement, and so on. Each of those parent statement types map to a binary feature.

AST Nesting Level. After some additional experimentation, we also decided to include the nesting level of a statement as an additional feature. This is the number of edges that must be traversed to reach the statement from the MethodDeclaration node in the AST. Intuitively, the reason for expecting that this feature might be useful is that statements nested deep inside other statements tend to handle low-level implementation details, rather than contributing to high-level program flow. We find that this feature is especially helpful when the goal is to reduce a method to a small fraction of its original length. For example, consider a method that parses a content string and generates a HostDirectives object containing the paths that are allowed to access by a software agent. The snippet is included in Appendix A.1 of the paper. The abridgment produced without the nesting level is given in Listing 10. The abridger chooses to retain code detailing the action to be taken if PATTERNS_USERAGENT is encountered, removing details regarding all of the other patterns that may be handled. If we include the nesting level as a feature, we obtain the abridgment of Listing 11, which retains information about all three of the patterns that the code handles.

Note that implementing this feature requires a bit of care. Rather than simply counting the depth of a node in the AST in every case, special rules must be generated on a case-by-case basis. For example, the Eclipse AST parser views each ElseIf branch as a child node nested under the preceding IfStatement. Rather than adding an additional depth level to each nested ElseIf, it makes more sense (and produces better results) if we treat all of the ElseIf clauses as occurring at the same level of nesting as the original IfStatement.

```
public static HostDirectives parse(String content,String myUserAgent){
      HostDirectives directives=null;
      boolean inMatchingUserAgent=false;
      StringTokenizer st=new StringTokenizer(content,"\n");
      while(st.hasMoreTokens()){
            String line=st.nextToken();...
            if(line.matches(PATTERNS_USERAGENT)){
                  String ua=line.substring(PATTERNS_USERAGENT_LENGTH).trim().toLowerCase();
                  if(ua.equals("*") || ua.contains(myUserAgent)){
                        inMatchingUserAgent=true;
                        if(directives == null){
                              directives=new HostDirectives();
                        }
                  } else {
                        inMatchingUserAgent=false;
                  }
            }else ...
      }
      return directives;
}
```

Listing 10. An abridgment removing two informative code branches.

```
public static HostDirectives parse(String content,String myUserAgent){
      HostDirectives directives=null;
      boolean inMatchingUserAgent=false;
      StringTokenizer st=new StringTokenizer(content,"\n");
      while(st.hasMoreTokens()){
            String line=st.nextToken();...
            if(line.matches(PATTERNS_USERAGENT)){
                  String ua=line.substring(PATTERNS_USERAGENT_LENGTH).trim().toLowerCase();
            } else if(line.matches(PATTERNS_DISALLOW)){...
                  String path=line.substring(PATTERNS_DISALLOW_LENGTH).trim();...
                  path=path.trim();...
            } else if(line.matches(PATTERNS_ALLOW)){...
                  String path=line.substring(PATTERNS_ALLOW_LENGTH).trim();...
                  path=path.trim();
                  directives.addAllow(path);
            }
      }
      return directives;
}
```

Listing 11. An abridgment obtained after including the nesting level.

Identifier Reference Count. This also turned out to be a useful feature. Imagine a statement in an arbitrary Java method. We find that by counting the number of identifiers (including variable accesses and function calls) that are used by this statement—and by descendent statements of this statement in AST—we obtain valuable information as to the importance of a statement. This tends to value complex statements that perform a lot of computation.

Text Classification. Besides features obtained from a static program analysis, we also decided to include the scores obtained from a simple text classifier as features for use in the abridgment. This is particularly useful in identifying statements that should *not* be included in the abridgment. For instance, consider the attribute name `onDebug` from line 6 of the `handleDeclaration` method of Listing 5. This indicates that the code has to do with debugging, and so the code associated with this statement may not be important for aiding a basic understanding of the program.

To use such text to power the abridgment, we first extract all of the literals and identifiers from each statement. For each, we assume the use of a camel or underscore naming convention, and split each identifier into multiple terms. For example, the method name `getHtmlParser` becomes three tokens: `get`, `html`, and `parser`. The variable name `TYPE_METHOD_REFERENCE` becomes the tokens `type`, `method`, and `reference`.

Perhaps the most natural way to include these terms in the abridgment procedure would be to simply include the presence/absence of each term as a feature in the IP formulation. The problem with this is that it could lead to many thousands of features, leading to a high training time for the optimization-within-optimization procedure. As an alternative, as a separate step before learning, we build a simple text classifier using the training data, that attempts to predict whether or not a statement is included in an abridgment based upon the terms that it contains. In our implementation, we use a naive Bayes classifier, though any other appropriate classifier could be used. This reduces the set of terms down to a single feature: a probability (predicted by the classifier) that a statement will be included in the abridged program, based solely on the textual terms contained within the code. The optimization-within-optimization procedure then learns a weight for for this probability.

5 EVALUATION

In this section, we describe a user study aimed at evaluating the abridgment algorithm described in the paper.

5.1 Study Goal

The core question we attempt to answer in this section is that: *Does use of the abridgment tool described in this paper allow engineers to understand source code more quickly, and/or with greater accuracy than can be achieved by simply perusing the original source code, or by making use of a straw-man abridgment tool?*

In order to evaluate the utility of our tool, we conduct an user study that empirically measures the speed and accuracy of code analysis achieved by engineers using the abridger that we have developed. We also include some survey questions that allow engineers to express their preferences and opinions of our tool.

5.2 Study Setup

Participants. The study participants consists of 23 graduate students and postdoctoral researchers (19 males and 4 females) in a computer science department. All are expert programmers, but they have different levels of Java competence. We ask each to estimate their own Java competence. Six evaluate themselves as expert, eleven evaluate themselves as proficient, three evaluate themselves as competent, two evaluate themselves as advanced beginners, and one evaluates himself/herself as a novice Java programmer.

Task Design. At a high level, participants in the study were given two tasks (plus a survey task described subsequently), which took the average participant around 36 minutes to complete (22 minutes for Task 1, and 14 minutes for Task 2):

In Task 1, participants are repeatedly given a natural language description of a desired Java method, and then they are asked to search through five different Java methods in order to identify the method described in English. Participants are evaluated both on the time required to complete the task, as well as their accuracy in completing the task.

This task simulates a situation where an engineer is looking for a particular example code, and has used a search engine to retrieve various results. The engineer needs to browse the results to determine which is most relevant. We are evaluating whether abridging the various code is useful in such a situation.

In detail, participants are first given a high-level description of the search-and-identify task. Then, participants are given two sample problems for practice. In one sample search problem, participants are given a set of non-abridged codes. In a second sample search problem, participants are given a set of abridged codes. After practice, participants are given the specific instructions, “Try to determine the correct code quickly as possible, while also ensuring that you have determined the correct code with ‘reasonable’ accurately, however you choose to define ‘reasonable’. If at some point you determine that it is impossible to locate the correct code, then simply choose the code that is your best guess as to the answer".

At this point, participants are given a series of six additional search problems, which are timed and where participants are tested for accuracy. Two of the problems are completed using unabridged code, two are completed using a naive abridger (see below) and two are completed using the abridger described in this paper. All six tasks are given in random order. Participants are aware of whether the code is original, abridged by method *A*, or abridged by method *B*, without being aware of any of the merits of the abridgment methods or the algorithms used. All abridged codes are shortened to 50% of the original length. We apply a constant compression rate instead of a constant raw number of lines since the same compression rate can be used regardless of the original code size, whereas the same raw number of lines after abridging makes less sense for codes of radically different lengths.

In Task 2, participants are repeatedly given pairs of Java methods, as well as two natural language descriptions, and they are asked to match the two descriptions with the codes. This task is similar to the first one, but since participants are given English descriptions of both of the codes and they are given only two codes in each particular problem, our expectation is that participants will read both of the codes much more carefully, as opposed to the skim-and-prune approach expectedly taken by most participants on the first task. This assumption is corroborated by the fact that participants spent 7 more seconds in average viewing each method in Task 2 than that of Task 1.

As before, participants are first given two sample problems for practice. Again, one sample problem uses original codes, and the second, abridged codes. After practice, participants are given the specific instructions, "Try to complete each matching as quickly as possible, while also ensuring that you have labeled each code with 'reasonable' accurately, however you choose to define 'reasonable'. If you determine that it is impossible to decide with more than 50/50 (coin-flip) accuracy which documentation goes with which method, then choose 'impossible to decide' as your answer".

After this, participants are given nine additional discrimination tasks, again in random order. Those nine tasks again evaluate each of the three display methods (original, abridged, and naive abridged).

A Straw-man Abridger. The naive abridgment method evaluated (the straw-man) is based upon our IP formulation, but it uses only the hard constraints introduced in our initial IP formulation, and it weights each statement type evenly. The abridgment generated by the straw-man can be viewed as a union of multiple revised program slices of the original code, since for each statement present in the abridged program, the control dependences and the revised data dependences (variable declaration dependences instead) are satisfied. The relative performance of participants using this straw-man abridgment allows us to determine whether it is enough to simply shorten a code, or if the exact nature of the abridgment matters.

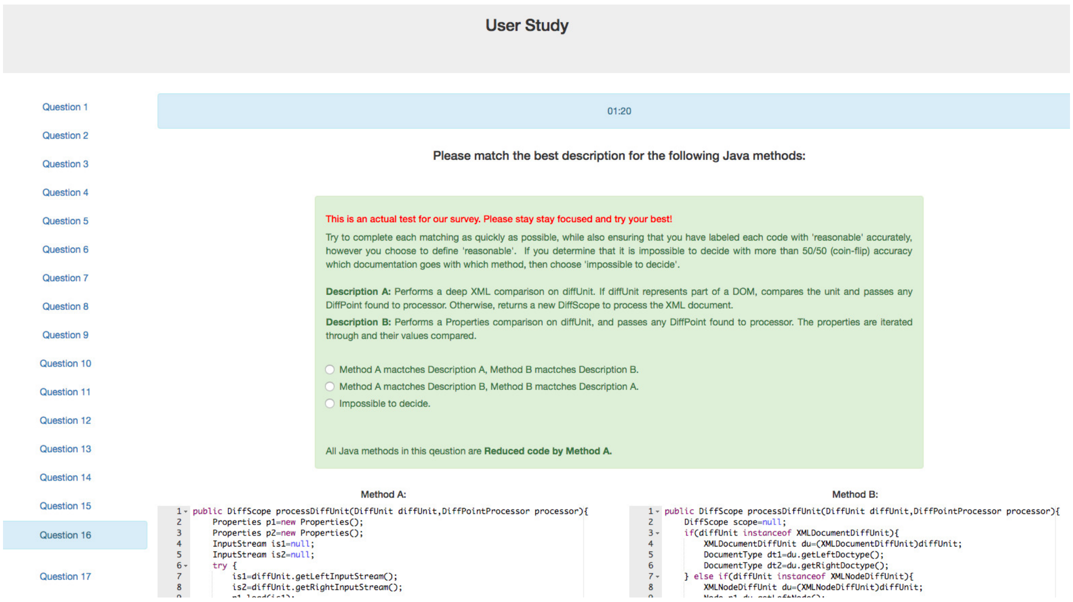
Sample Program Selection. To make our study more realistic, we ensure that each set of five methods for each problem from Task 1, and each pair of methods for each problem from Task 2, are similar—it is not interesting, for example, to ask a Task 1 participant to determine which method is implementing an `SMTP` protocol connection when the other four methods are computer graphics code.

Thus, we use the Sourcerer maven-src dataset [Lopes et al. 2010] to supply the Java methods used in the study. We select programs from six different maven-src category labels for our study: *Diff and Patch Libraries*, *Web Testing*, *Mail Clients*, *XPath Libraries*, *HTML Parser*, and *OAuth Libraries*. To create a particular problem, we first search one of the six maven-src categories using a set of reasonable keywords. For example, we search the category *Mail Clients* using the keywords "smtp, connection" for similar methods that each have to do with SMTP protocol connections. We select methods across problems so that they have a variety of different lengths: 30, 50, 80, 120 lines of code, for example. In this way, Task 1 consists of a total of (6 problems + 2 warm-ups) ×5 codes per problem = 40 methods. Task 2 consists of a total of (9 problems + 2 warm-ups) ×2 codes per problem = 22 methods. Each participant sees exactly this set of 62 methods, but the 48 non-warmup methods are randomly assigned to be either abridged, abridged using the naive method, or a full code. In this way, we do not bias our results by, for example, abridging only the easiest codes. Note that there is no overlap between the set of methods included in the user study and our training set.

All English descriptions used in the study are directly abstracted from the methods' original Javadoc comments.

Our current implementation only considers the statements nested in a method body when abridging. Other information (such as Javadoc comments) may be useful, but this is left for future work.

Other Details. The user study was deployed as a web application and participants were invited to complete the study at their leisure, wherever they liked, though they we asked to complete the study in a single sitting. A screen shot of the web application for Task 2 is illustrated in Figure 1. An example Java method that appeared in Task 2 is included in Appendix A.2 along with the natural language description given to participants, as well as both of the abridgments.



Please match the best description for the following Java methods:

This is an actual test for our survey. Please stay stay focused and try your best!

Try to complete each matching as quickly as possible, while also ensuring that you have labeled each code with 'reasonable' accurately, however you choose to define 'reasonable'. If you determine that it is impossible to decide with more than 50/50 (coin-flip) accuracy which documentation goes with which method, then choose 'impossible to decide'.

Description A: Performs a deep XML comparison on diffUnit. If diffUnit represents part of a DOM, compares the unit and passes any DiffPoint found to processor. Otherwise, returns a new DiffScope to process the XML document.

Description B: Performs a Properties comparison on diffUnit, and passes any DiffPoint found to processor. The properties are iterated through and their values compared.

- ○ Method A mactches Description A, Method B mactches Description B.
- ○ Method A mactches Description B, Method B mactches Description A.
- ○ Impossible to decide.

All Java methods in this qeustion are **Reduced code by Method A.**

Method A:

```java
public DiffScope processDiffUnit(DiffUnit diffUnit,DiffPointProcessor processor){
    Properties p1=new Properties();
    Properties p2=new Properties();
    InputStream is1=null;
    InputStream is2=null;
    try {
        is1=diffUnit.getLeftInputStream();
        is2=diffUnit.getRightInputStream();
```

Method B:

```java
public DiffScope processDiffUnit(DiffUnit diffUnit,DiffPointProcessor processor){
    DiffScope scope=null;
    if(diffUnit instanceof XMLDocumentDiffUnit){
        XMLDocumentDiffUnit du=(XMLDocumentDiffUnit)diffUnit;
        DocumentType dt1=du.getLeftDoctype();
        DocumentType dt2=du.getRightDoctype();
    } else if(diffUnit instanceof XMLNodeDiffUnit){
        XMLNodeDiffUnit du=(XMLNodeDiffUnit)diffUnit;
```

Fig. 1. A screen shot of our web page that participants use to conduct the survey for Task 2.

5.3 Results and Statistical Analysis

Raw results are given in Tables 1 through 4. While it would seem that there are significant differences among the various approaches, we also wanted to conduct a proper statistical analysis, as we describe now.

Null Hypotheses. To do this, we define eight different null hypotheses.

Table 1. Number of correct/incorrect responses for the six search problems of Task 1.

	Original Code				Naive Approach				Final Approach			
Prob	Cor	Incor	Tot	Rate	Cor	Incor	Tot	Rate	Cor	Incor	Tot	Rate
1	6	1	7	0.857	3	5	8	0.375	6	2	8	0.750
2	4	4	8	0.500	3	4	7	0.429	6	2	8	0.750
3	5	3	8	0.625	6	2	8	0.750	6	1	7	0.857
4	7	0	7	1.00	8	0	8	1.00	8	0	8	1.00
5	4	4	8	0.500	5	2	7	0.714	8	0	8	1.00
6	4	4	8	0.500	5	3	8	0.625	6	1	7	0.857
Total	30	16	46	0.652	30	16	46	0.652	40	6	46	0.870

The four null hypotheses $H_0^{1,\text{time},\text{orig}}$, $H_0^{1,\text{acc},\text{orig}}$, $H_0^{1,\text{time},\text{naive}}$, $H_0^{1,\text{acc},\text{naive}}$ correspond to Task 1. "Time" refers to a null hypothesis having to do with completion time, "acc" having to do with accuracy, "orig" is a null hypothesis comparing the full abridgment method with the original code, and "naive" compares the full abridgment method with the naive abridgment. Hence, for example, $H_0^{1,\text{time},\text{naive}}$ corresponds to the assertion "For Task 1, the expected time to complete the search task using the full abridgment is not any less than the expected time using the naive approach." And, for another example, $H_0^{1,\text{acc},\text{orig}}$ corresponds to the assertion "For Task 1, the expected accuracy with which one completes the search task using the full abridgment is not any better than the expected accuracy using the original code."

Similarly, the four null hypotheses $H_0^{2,\text{time},\text{orig}}$, $H_0^{2,\text{acc},\text{orig}}$, $H_0^{2,\text{time},\text{naive}}$, $H_0^{2,\text{acc},\text{naive}}$ correspond to Task 2.

The Bootstrap. Checking whether the observed data tends to refute each hypothesis requires a statistical test of significance capable of obtaining a p-value for each. Unfortunately, a classical test such as a t-test seems inappropriate, since when we test a particular hypothesis, we have multiple sources of correlations across the various measurements. For example, when looking at a specific problem, the observed measurement (accuracy or time of completion) is likely not only related to the participants' expertise (a source of correlation that is in fact handled by the paired test) but also the characteristics of the codes tested for that problem.

To address this, we decide to utilize bootstrap [Efron 1982], a simulation-based significance computation that naturally takes into account of such issues. The general idea behind bootstrap is to simulate a large number of data sets from the original data set by re-sampling with replacement. The null hypothesis is checked on each simulated data set, and the p-value is approximated by the fraction of the time that the null hypothesis holds.

In our scenario, we generate a simulated data set as follows. Given the original participants, we first re-sample from the set of participants, with replacement. Then for each re-sampled participant, we re-sample the set of problems that were given to that participant, with replacement as well. The null hypothesis in question is tested on each of $100,000$ simulated data sets, and the number of times that it holds is used as an estimate of the p-value at which the hypothesis can be rejected.

The approximated p-value for each hypothesis is listed in Table 5.

5.4 Discussion

The results obtained seem to show both the practical and the statistical significance of the proposed abridgment methodology. Consider Table 1, which details the accuracy results obtained on the code search problem. While the unabridged code and the naive abridgment both result in around 65% accuracy, the full IP formulation abridgment results in 87% accuracy, effectively reducing the error rate by more than $\frac{2}{3}$, from 35% to just 13%. Likewise, the time average time required to answer the

Table 2. Average time required (in seconds) for participants to complete each Task 1 search problem.

Problem no.	Original Code	Naive Approach	Final Approach
1	162	223	229
2	254	212	197
3	182	118	134
4	111	84	126
5	200	127	93
6	138	148	83
Average	135	116	83

Table 3. Number of correct/incorrect responses for the nine labeling problems of Task 2.

	Original Code				Naive Approach				Final Approach			
Prob	Cor	Incor	Tot	Rate	Cor	Incor	Tot	Rate	Cor	Incor	Tot	Rate
1	8	0	8	1.00	5	2	7	0.714	8	0	8	1.00
2	7	1	8	0.875	5	3	8	0.625	6	1	7	0.857
3	5	2	7	0.714	6	2	8	0.750	6	2	8	0.750
4	5	3	8	0.625	5	2	7	0.714	4	4	8	0.500
5	4	4	8	0.500	7	1	8	0.875	6	1	7	0.857
6	5	2	7	0.714	8	0	8	1.00	8	0	8	1.00
7	6	2	8	0.750	2	5	7	0.286	6	2	8	0.750
8	5	3	8	0.625	5	3	8	0.625	7	0	7	1.00
9	5	2	7	0.714	7	1	8	0.875	7	1	8	0.875
Total	50	19	69	0.725	50	19	69	0.725	58	11	69	0.841

Table 4. Average time required (in seconds) for participants to complete each Task 2 labeling problem.

Problem no	Original Code	Naive Approach	Final Approach
1	83	52	68
2	71	77	44
3	85	76	72
4	118	80	117
5	133	63	54
6	74	66	84
7	90	96	58
8	104	85	51
9	53	54	58
Average	63	49	40

Table 5. P-value for hypothesis test.

Null Hyp.	p-Value	Null Hyp.	p-Value	Null Hyp.	p-Value	Null Hyp.	p-Value
$H_0^{1,\text{acc},\text{orig}}$	0.015	$H_0^{1,\text{time},\text{orig}}$	0.0392	$H_0^{1,\text{acc},\text{naive}}$	0.0185	$H_0^{1,\text{time},\text{naive}}$	0.3807
$H_0^{2,\text{acc},\text{orig}}$	0.082	$H_0^{2,\text{time},\text{orig}}$	0.0077	$H_0^{2,\text{acc},\text{naive}}$	0.0899	$H_0^{2,\text{time},\text{naive}}$	0.3134

question drops from 135 seconds (using the original code) all the way down to 83 seconds. This is a good increase in accuracy, at the same time that the time required for a human to process the code is decreased by 39%.

Similar results are obtained for Task 2, though the difference is not as striking, perhaps because the task is somewhat less well-suited to abridgment (scanning a list of codes to match an English description, as in Task 1, would seem to be the ideal case). Again, both the original code and the

Table 6. Survey of participants' feedback.

1. Which version of the code tended to make it easier to write your description?					
Non-abridged	6	Naive	6	Final Approach	11
2. Which abridgment level is easiest to use for the naive approach?					
To around 15 lines	9	To around 30 lines	5	To 50%	9
3. Which abridgment level is easiest to use in conjunction with the full method?					
To around 15 lines	11	To around 30 lines	3	To 50%	9
4. Which abridgment removes the most statements necessary understand the code's high-level functionality?					
Naive Approach		17	Final Approach		6
5. Which abridgment contains the most unnecessary statements for understanding the code's high-level functionality?					
Naive Approach		14	Final Approach		9

naive abridgment result in the same average accuracy: 73%. However, the full IP formulation results in 84% accuracy. Again, the error rate has been decreased significantly, this time from 27% to 16%. This is while allowing the average participant to decrease the average problem solution time from 63 seconds all the way down to 40 seconds.

While all of the results seem to show practical significance, the only results that do not seem to show statistical significance are the comparison of participant task completion speeds, comparing the two abridgment methods. This is perhaps not surprising, if one assumes that the time required for a human to "process" a code is proportional to the length of the code.

The strongest null hypothesis rejections occurred when looking at the accuracy numbers on Task 1, both of which allow a rejection at a p-value of less than 0.02. It is interesting to note that there is a bit of a discrepancy between the statistical significance and the practical significance of the results, with the former generally lagging behind the latter. The explanation for this is the fact that we only had 23 study participants—extending these results with a larger study would be an important problem for future work.

That said, we would assert that taken in their totality, these results seem to argue for the practical utility of the proposed abridgment methodology for a variety of software engineering tasks. The abridgments both increase human efficiency while decreasing human error rates when examining code.

5.5 Subjective Evaluation

We also conduct a subjective survey to collect participants' opinions of the method.

Setup and Results. To collect participants' opinions, each of the 23 study participants was provided with five Java methods of 60 to 120 lines. For each method, a non-abridged code, an abridged code using the naive method, and an abridged code using the final approach are provided. For the abridged codes, participants can also change the aggressiveness of the abridgment ("To around 15 lines", "To around 30 lines", or "To 50% of the original lines"). We ask participants to peruse the various options and then tell them, "Try to write a one-sentence description of the method. This description should inform someone at a high level what the method does. Try to write your description quickly as possible, while also ensuring that your description is of 'reasonable' quality, however you define 'reasonable'". This ensures that each participant considered each of the codes and abridgment methods carefully.

We then ask each participant the questions listed in Table 6; the results are listed there as well. Note, however, that participants are not aware of which abridger is the "naive" abridger, and which is the one described in this paper.

Discussion. At a high level, these results seem to echo the observed, quantitative results, though the qualitative opinions seemed to be more evenly divided compared to the quantitative results, which showed a clear and significant benefit compared to the proposed abridgments. For example, 74% of the participants preferred some version of abridgment to using the full code to write a summary, and of those who preferred an abridgment, 65% preferred the full IP abridgment. These are relatively clear results in favor of abridgment, but far less striking than the quantitative results. Participants are more strongly in favor of the full solution when they are directly asked which abridgment removes too many useful program statements: 74% of participants thought that the naive abridgment was inferior in this regard. But these results, while compelling, are far less significant than the quantitative results.

5.6 Threats to Validity

While the experimental results seem to support the hypothesis that the proposed abridgment system is useful, there are several reasons to be at least a bit cautious in asserting that these results will extend past a controlled laboratory setting and to the "real world".

First, we re-emphasize that the study was performed in a controlled setting where it was likely fairly obvious that we were trying to study the effect of abridgment on a programmer's ability to quickly and effectively understand code. Since we were obviously studying abridgment, participants could have been especially motivated to prove how well they could quickly analyze the abridged code, which may have biased results.

Likewise, the tasks we devised, which involved asking participants to quickly match English descriptions to codes, are merely a proxy for a real-world setting where an engineer uses a code search tool that returns a number of codes that the user must peruse. In a real-world setting, an engineer will not be scanning codes to answer textual questions (as we asked participants to do) s/he will be scanning codes to find a help in solving a particular engineering task. There is a natural concern that the questions we asked were somehow biased or not otherwise representative of the target, code-search task. Further, even if our results hold in the target environment, it may be that other factors—such as the order in which the results are presented—are far more important than abridgment in determining the utility of a code search tool.

It is also important to point out that while many of our results reached statistical significance, this was with respect to our study population, which was a group of expert programmers in a computer science department. This group may not be representative of programmers in a typical engineering setting. This group is overwhelmingly male (83% male, representing a typical postdoctoral and graduate student population at a university). Further, while there were varying levels of Java expertise, all participants were expert programmers, many with PhD degrees in computer science. But while all were expert programmers, not all were expert Java programs. This may or may not be representative of a typical user group for an abridger.

6 RELATED WORK

Program slicing is a technique to identify a subset of statements in a program that may influence the values relevant to a seed statement, noted as a slicing criterion. A static program slice consists of all statements in program that may affect the value of variables at the point where the seed statement executes [Weiser 1981]. A dynamic program slice consists of all statements that actually affect the value of a variable occurrence for a given program input [Agrawal and Horgan 1990]. Programmers can benefit from the technique to remove irrelevant details in a program during code inspection and debugging. Sometimes a program slice is still too large for an engineer to consume, and not all statements in the slice appear equally relevant regarding to a program understanding task. To address this difficulty, Sridharan et al. [2007] proposed the concept of thin slice, which only consists of producer statements for the slicing criterion. The producer statements help compute and copy a value to the seed statement. On the other hand, statements that explain why producers affect the seed are excluded. Their study illustrated significant reduction of the retained statements. Nevertheless, an obvious issue of applying program slicing to abridge code is that users have to explicitly specify the slicing criterion. Based on our knowledge, there is limited effort trying to adaptively determine the slice criterion for program comprehension tasks.

Natural language summarization techniques have been studied extensively [Jones 2007]. The most representative approach is to extract the most relevant text segments from documents [Fung et al. 2003; Kikuchi et al. 2003; Wong et al. 2008]. Recently, automatic summarization techniques for source code also draw attention of researchers from the software engineering research community. For example, Haiduc et al. [2010b] applies text retrieval techniques to generate summaries; Eddy et al. [2013], McBurney et al. [2014], and Liu et al. [2014] improve the quality of the summary by considering topic modeling. More specifically, technologies to automatically generate descriptive comments for exceptions thrown by Java methods [Buse and Weimer 2008], parameters of Java methods [Sridhara et al. 2011b], failed tests [Zhang et al. 2011], Java methods [Sridhara et al. 2010, 2011a], Java classes [Moreno 2014; Moreno et al. 2013], Java method context [McBurney and McMillan 2014, 2016], release notes and specific maintenance tasks [Moreno 2014] are proposed. These approaches try to generate descriptive documentation for source code at different levels. Usually, the methods to locate key words for natural language generators are based on different heuristics. For instance, Sridhara et al. [2011a] proposes algorithms to identify sequential, conditional and looping code fragments that can be handled by their natural language synthesizers. As their evaluation reveals, the frequency of the identified code fragments ranges from 11% to 40%. In other words, there are plenty of cases that the proposed method cannot handle. Similarly, Moreno et al. [2013] applies a technique to determine method and class stereotypes [Dragan et al. 2006, 2010; Moreno and Marcus 2012], then uses heuristics based stereotypes to select statements for summary generation. However, method/class stereotype cannot elaborate all features of the source code. Important implementation outlines may also be missed by this approach.

Auto folding [Fowkes et al. 2014] is another tool to aid program comprehension. This tool adapts TopicSum [Haghighi and Vanderwende 2009], a scoped topic model that extends the latent Dirichlet allocation (LDA) model [Blei 2012; Blei et al. 2003] to handle topics at multiple levels to Java source code by defining topics for each file, topics for each software project, and background topics. The auto folding result is generated by an iterative greedy optimization algorithm to extract the most relevant rooted contiguous subtree from the foldable tree given constraints on the subtree size. Applying hierarchical topics is novel and insightful. However, not all source code fragments are organized with file and project, e.g., source code fragments attached with answers on Stack Overflow. Additionally, the assumption to handle folding by block is intuitive but not essential. We believe a more reasonable basic element for program comprehension is a single statement.

Ying and Robillard [2013] also applies supervised learning technique to generate partial programs that serve the purpose of demonstrating the usage of an API. A binary classifier is trained to predict whether a line in a code fragment should be included in the partial program. Since this approach focuses on selecting code snippets related to a particular API usage, the classifier makes the prediction for each line independently without considering the context for that line to be executed. In contrast, code abridgment targets selecting a subset of statements in the original program in order to help users understand the overall functionality of the code. Code abridgment not only includes features for each statement, but also considers complicated dependences between statements, which makes the formulation of the problem and the training process more challenging.

7 CONCLUSION AND FUTURE WORK

We have presented a novel approach that automatically abridges a Java method in order to display the code in a small space without compromising a reader's ability to understand the basic functionality of the code. We have applied a supervised machine learning approach, which allows our algorithm to mimic the human abridger by learning from the human-created code abridgment. We conducted a comprehensive user study, which provided evidence that an automatically truncated code can actually result in a greater level of understanding in less time than the original code.

Since our stated goal is to perform abridgment for displaying code search results, one obvious avenue for future work would be to couple our abridgment tool with an actual code search tool and then evaluate the utility of the combined tool for use in various software engineering tasks. The utility of a combined tool could be compared against code search without abridgment. This more direct evaluation would give a much clearer picture as to whether abridgment is a crucial part of code search, and would alleviate some of the concerns detailed under "Threats to Validity" in a previous section of the paper.

One important question that we did not consider in the paper is how to determine the correct size of an abridgment. Even basic questions along these lines remain unanswered. Does the proportion of code retained matter? Or is it the absolute number of lines that is important? In either case, what is the correct proportion/number? These are important questions for follow-up work.

There are also many ways that we could consider to improve the abridgments themselves. In this paper, we have not taken into account input to a code search task when abridging the code. Tailoring an abridgment to a particular search query might make sense. Aside from search query input, there are many other features that we could take into account during code search to perform abridgment: repository change history, user search history and feedback in the form of previous user click-streams, and even contextual information such as details regarding the particular engineering task that the user was working on when the search was performed (including the code that was most recently written, the APIs being used, and caller and callee information). Another way to improve the quality of the abridgment may be to use different optimization procedures. For example, rather than using an IP formulation, a genetic or evolutionary algorithm may make sense. All of these could improve the abridgment results.

A APPENDIX

A.1 Code Snippet Mentioned in Section 4

```java
public static HostDirectives parse(String content,String myUserAgent){
      HostDirectives directives=null;
      boolean inMatchingUserAgent=false;
      StringTokenizer st=new StringTokenizer(content,"\n");
      while(st.hasMoreTokens()){
            String line=st.nextToken();
            int commentIndex=line.indexOf("#");
            if(commentIndex > -1){
                  line=line.substring(0,commentIndex);
            }
            line=line.replaceAll("<[^>]+>","");
            line=line.trim();
            if(line.length() == 0){
                  continue;
            }
            if(line.matches(PATTERNS_USERAGENT)){
                  String ua=line.substring(PATTERNS_USERAGENT_LENGTH).trim().toLowerCase();
                  if(ua.equals("*") || ua.contains(myUserAgent)){
                        inMatchingUserAgent=true;
                        if(directives == null){
                              directives=new HostDirectives();
                        }
                  } else {
                        inMatchingUserAgent=false;
                  }
            }else if(line.matches(PATTERNS_DISALLOW)){
                  if(!inMatchingUserAgent){
                        continue;
                  }
                  String path=line.substring(PATTERNS_DISALLOW_LENGTH).trim();
                  if(path.endsWith("*")){
                        path=path.substring(0,path.length() - 1);
                  }
                  path=path.trim();
                  if(path.length() > 0){
                        directives.addDisallow(path);
                  }
            }else if(line.matches(PATTERNS_ALLOW)){
                  if(!inMatchingUserAgent){
                        continue;
                  }
                  String path=line.substring(PATTERNS_ALLOW_LENGTH).trim();
                  if(path.endsWith("*")){
                        path=path.substring(0,path.length() - 1);
                  }
                  path=path.trim();
                  directives.addAllow(path);
            }
      }
      return directives;
}
```

Listing 12. A Java method that parses a content string and generate a object containing the paths that are allowed to access by the specified agent.

A.2 A Code Example in the User Study

This is a Java method appears in Task 2 of our user study. The description for this method is that "submit the current form with the specified submit button." The original method, the abridgment produced by the naive approach, and the abridgment produced by our final approach are shown in Listing 13 through Listing 15.

```java
public void submit(String buttonName,String buttonValue){
      List<HtmlElement> l=new LinkedList<HtmlElement>();
      l.addAll(getForm().getInputsByName(buttonName));
      l.addAll(getForm().getButtonsByName(buttonName));
      try {
            for (int i=0; i < l.size(); i++){
                  Object o=l.get(i);
                  if(o instanceof HtmlSubmitInput){
                        HtmlSubmitInput inpt=(HtmlSubmitInput)o;
                        if(inpt.getValueAttribute().equals(buttonValue)){
                              inpt.click();
                              return;
                        }
                  }
                  if(o instanceof HtmlImageInput){
                        HtmlImageInput inpt=(HtmlImageInput)o;
                        if(inpt.getValueAttribute().equals(buttonValue)){
                              inpt.click();
                              return;
                        }
                  }
                  if(o instanceof HtmlButton){
                        HtmlButton inpt=(HtmlButton)o;
                        if(inpt.getTypeAttribute().equals("submit") && inpt.getValueAttribute().equals(buttonValue)){
                              inpt.click();
                              return;
                        }
                  }
            }
      } catch (FailingHttpStatusCodeException e){
            if(!ignoreFailingStatusCodes){
                  throw new TestingEngineResponseException(e.getStatusCode(),e);
            }
            return;
      } catch (IOException e){
            throw new RuntimeException("Html Error using submit button with ["+buttonName+"] and ["+buttonValue+"]",e);
      }
      throw new RuntimeException("No submit button in current form with ["+buttonName+"] and ["+buttonValue+"]");
}
```

Listing 13. The `submit` Java method.

```java
public void submit(String buttonName,String buttonValue){
      List<HtmlElement> l=new LinkedList<HtmlElement>();
      l.addAll(getForm().getInputsByName(buttonName));
      l.addAll(getForm().getButtonsByName(buttonName));
      try {
            for (int i=0; i < l.size(); i++){
                  Object o=l.get(i);
                  if(o instanceof HtmlSubmitInput){
                        HtmlSubmitInput inpt=(HtmlSubmitInput)o;
                        if(inpt.getValueAttribute().equals(buttonValue)){
                              inpt.click();
                              return;
                        }
                  }...
            }
      } catch (FailingHttpStatusCodeException e){...
            return;
      } catch (IOException e){
            throw new RuntimeException("Html Error using submit button with ["+buttonName+"] and ["+buttonValue+"]",e);
      } ...
}
```

Listing 14. An abridgment of the `submit` method to around 50% of the original length generated by the naive approach.

```java
public void submit(String buttonName,String buttonValue){
      List<HtmlElement> l=new LinkedList<HtmlElement>();
      l.addAll(getForm().getInputsByName(buttonName));
      l.addAll(getForm().getButtonsByName(buttonName));
      try {
            for (int i=0; i < l.size(); i++){
                  Object o=l.get(i);
                  if(o instanceof HtmlSubmitInput){
                        HtmlSubmitInput inpt=(HtmlSubmitInput)o;...
                  }
                  if(o instanceof HtmlImageInput){
                        HtmlImageInput inpt=(HtmlImageInput)o;...
                  }
                  if(o instanceof HtmlButton){...
                        if(inpt.getTypeAttribute().equals("submit") && inpt.getValueAttribute().equals(buttonValue)){...
                              return;
                        }
                  }
            }
      } catch (FailingHttpStatusCodeException e){...
      } catch (IOException e){...
      } ...
}
```

Listing 15. An abridgment of the `submit` method to around 50% of the original length generated by our final approach.

ACKNOWLEDGEMENTS

Material in this paper has been supported by the NSF under grant numbers 1355998 and 1409543, and by the DARPA MUSE program, award #FA8750-14-2-0270.

REFERENCES

Nahla J Abid, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2015. Using stereotypes in the automatic generation of natural language summaries for c++ methods. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 561–565.

Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPlan Notices*, Vol. 25. ACM, 246–256.

David M Blei. 2012. Probabilistic topic models. *Commun. ACM* 55, 4 (2012), 77–84.

David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

Raymond PL Buse and Westley R Weimer. 2008. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 273–282.

Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2006. Reverse engineering method stereotypes. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 24–34.

Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2010. Automatic identification of class stereotypes. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–10.

Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 13–22.

Bradley Efron. 1982. *The jackknife, the bootstrap and other resampling plans*. SIAM.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

Jaroslav Fowkes, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2014. Autofolding for Source Code Summarization. *arXiv preprint arXiv:1403.4503* (2014).

Pascale Fung, Grace Ngai, and Chi-Shun Cheung. 2003. Combining optimal clustering and hidden Markov models for extractive summarization. In *Proceedings of the ACL 2003 workshop on Multilingual summarization and question answering-Volume 12*. Association for Computational Linguistics, 21–28.

Aria Haghighi and Lucy Vanderwende. 2009. Exploring content models for multi-document summarization. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 362–370.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010a. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 223–226.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010b. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 35–44.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36, 1 (2009), 267–306.

Frank Hutter, Holger H Hoos, and Thomas Stützle. 2007. Automatic algorithm configuration based on local search. In *AAAI*, Vol. 7. 1152–1157.

Karen Spärck Jones. 2007. Automatic summarising: The state of the art. *Information Processing & Management* 43, 6 (2007), 1449–1481.

Tomonori Kikuchi, Sadaoki Furui, and Chiori Hori. 2003. Automatic speech summarization based on sentence extraction and compaction. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, Vol. 1. IEEE, I–I.

Yu Liu, Xiaobing Sun, Xiangyue Liu, and Yun Li. 2014. Supporting program comprehension with program summarization. In *Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference on*. IEEE, 363–368.

C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI Source Code Data Sets. (2010). http://www.ics.uci.edu/$\sim$lopes/datasets/

Paul W McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. 2014. Improving topic model source code summarization. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 291–294.

Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 279–290.

Paul W McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

Laura Moreno. 2014. Summarization of complex software artifacts. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 654–657.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

Laura Moreno and Andrian Marcus. 2012. Jstereocode: automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 358–361.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011a. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 101–110.

Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011b. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 71–80.

Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. *ACM SIGPLAN Notices* 42, 6 (2007), 112–122.

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

Kam-Fai Wong, Mingli Wu, and Wenjie Li. 2008. Extractive summarization using supervised and semi-supervised learning. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*. Association for Computational Linguistics, 985–992.

Annie TT Ying and Martin P Robillard. 2013. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 655–658.

Sai Zhang, Cheng Zhang, and Michael D Ernst. 2011. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 63–72.