Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models

Joshua Charles Campbell
Department of Computing Science
University of Alberta
Edmonton, Canada
joshua2@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
hindle1@ualberta.ca

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Canada
amaral@cs.ualberta.ca

ABSTRACT

A frustrating aspect of software development is that compiler error messages often fail to locate the actual cause of a syntax error. An errant semicolon or brace can result in many errors reported throughout the file. We seek to find the actual source of these syntax errors by relying on the consistency of software: valid source code is usually repetitive and unsurprising. We exploit this consistency by constructing a simple N-gram language model of lexed source code tokens. We implemented an automatic Java syntax-error locator using the corpus of the project itself and evaluated its performance on mutated source code from several projects. Our tool, trained on the past versions of a project, can effectively augment the syntax error locations produced by the native compiler. Thus we provide a methodology and tool that exploits the naturalness of software source code to detect syntax errors alongside the parser.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, diagnostics*; D.3.8 [**Programming Languages**]: Processors—*Parsing*

General Terms

Languages, Human Factors

Keywords

naturalness, language, n-grams, syntax, error location, NLP

1. MOTIVATION

Syntax errors plague new programmers as they struggle to learn computer programming [16, 4]. Even experienced programmers get frustrated by syntax errors, often resorting to erratically commenting their code out in the hope that they discover the location of the error that brought their development to a screeching halt [14]. Syntax errors are annoying to programmers, and sometimes very hard to find.

Garner et al. [4] admit that "students very persistently keep seeking assistance for problems with basic syntactic details." They, corroborated by numerous other studies [11, 12, 10, 17], found that errors related to basic mechanics (semicolons, braces, etc.) were the most persistent and common problems that beginner programmers faced. In fact these errors made up 10% to 20% of novice programmer compiler problems. As an example, consider the missing brace at the end of line 2 in the following Java code taken from the Lucene 4.0.0 release:

```
1 for (int i = 0; i < scorers.length; i++) {
2     if (scorers[i].nextDoc() == NO_MORE_DOCS)
5         lastDoc = NO_MORE_DOCS;
6         return;
7     }
8 }
```

This mistake, while easy for an experienced programmer to understand and fix, if they know where to look, causes the Oracle Java compiler [1] to report 50 error messages, including those in Figure 1, none of which are on the line with the mistake. This poor error reporting slows down the software development process because a human programmer must examine the source file to locate the error, which can be a very time consuming process.

Some smart IDEs such as Eclipse, aim to address this problem but still fall short of locating the actual cause of the syntax error as seen in Figure 2. Thus syntax errors and poor compiler/interpreter error messages have been found to be a major problem for inexperienced programmers [16, 4, 17].

There has been much work on trying to improve error location detection and error reporting. Previous methods tend to rely on rules and heuristics [3, 6, 8]. In contrast to those methods, we seek to improve on error location reporting by exploiting recent research on the regularity and naturalness of written source code [7]. Recent research by Hindle *et al.* [7] exploits n-gram based language models, applied to existing source code, to predict tokens for code completion.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597102

[1] Oracle's Java compiler, version 1.7.0_13, is available at `http://www.oracle.com/technetwork/java/javase/downloads/index.htm`

```
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:56:
 error: <identifier> expected
    ArrayUtil.mergeSort(scorers, new Comparator<Scorer>() { // sort the array
                        ^
                                                    ... 48 errors omitted ...
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:152:
error: class, interface, or enum expected
  }
  ^
[javac] 50 errors
```

Figure 1: Oracle Java Error messages from the motivational example.

The effectiveness of the n-gram model at code completion and suggestion is due to the repetitive, and consistent, structure of code.

Using this exact same model, and exploiting the natural structure of source code, we can improve error location detection. The idea behind this new method is to train a model on compilable source-code token sequences, and then evaluate on new code to see how often those sequences occur within the model. Source code that does not compile should be surprising for an n-gram language model trained on source code that compiles. Intuitively, most available and distributed source code compiles. Projects and their source code are rarely released with syntax errors, although this property may depend on the project's software development process. Thus, existing software can act as a corpus of compilable and working software. Furthermore, whenever a source file successfully compiles it can automatically update the training corpus because the goal is to augment the compiler's error messages. Changes to the software might not compile. When that happens, locations in the source file that the model finds surprising should be prioritized as locations that a software engineer should examine.

The following sections of the paper examine the feasibility of this new method by testing its ability to locate errors in a variety of situations, as an augmentation of compiler error reporting. These situations include situations which are much more adverse than expected in practice.

The main contributions of this work include:

- A method of statistical, probabilistic syntax-error location detection that exploits n-gram language models.
- A prototype implementation of an n-gram language-model-based Java syntax error locator, UnnaturalCode,[2] that can be used with existing build systems and Java compilers to suggest locations that might contain syntax errors.
- A validation of the feasibility of the new syntax error location detection method.
- A validation of the integration of the new method with the compiler's own methods.
- A modified version of `MITLM`,[3] which has routines developed by the authors to calculate the entropy of short sentences with respect to a large corpus quickly.

[2]UnnaturalCode is available at `https://github.com/orezpraw/unnaturalcode`

[3]The modified `MITLM` package used in this paper is available at `https://github.com/orezpraw/MIT-Language-Modeling-Toolkit`

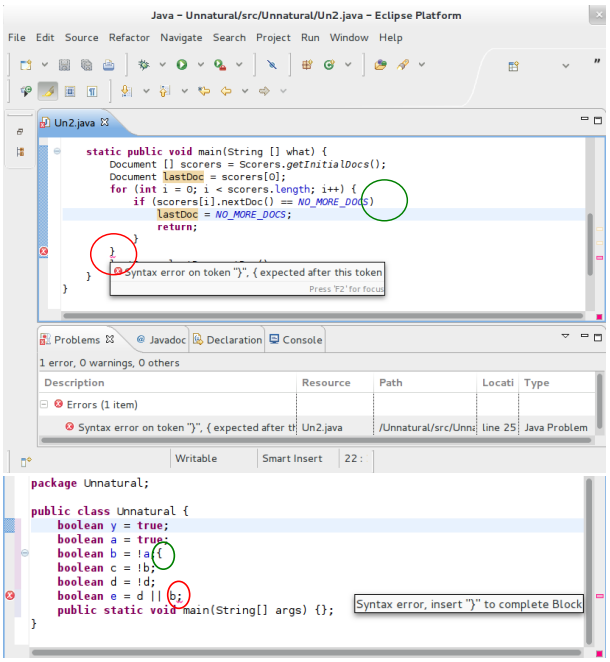

Figure 2: Top: Figure 1's code snippet put into Eclipse; bottom: another example of failed Eclipse syntax error location detection. Eclipse often detects syntax errors, but often reports them in the wrong location. Eclipse's suggested error location is circled in red, and the actual error location is circled in green.

2. BACKGROUND

An n-gram language model at its lowest level is simply a collection of counts. These counts represent the number of times a phrase appears in a corpus. These phrases are referred to as n-grams because they consist of at most n words or tokens. These counts are then used to infer the probability of a phrase: the probability is simply the frequency of occurrence of the phrase in the original corpus. For example, if the phrase "I'm a little teapot" occurred 7 times in a corpus consisting of 700 4-grams, its probability would be .01. However, it is more useful to consider the probability of a word in its surrounding context. The probability of finding the word "little" given the context of "I'm a _____ teapot" is much higher because "little" may be the only word that shows up in that context — a probability of 1.

These n-gram models become more accurate as n increases because they can count longer, more specific phrases. However, this relationship between n and accuracy is also problematic because most n-grams will not exist in a corpus of

human-generated text (or code). Therefore most n-grams would have a probability of zero for a large enough n. This issue can be addressed by using a smoothed model. Smoothing increases the accuracy of the model by estimating the probability of unseen n-grams from the probabilities of the largest m-grams (where $m < n$) that the n-gram consists of and that exist in the corpus. For example, if the corpus does not contain the phrase "I'm a little teapot" but it does contain the phrases "I'm a" and "little teapot" it would estimate the probability of "I'm a little teapot" using a function of the two probabilities it does know. In UnnaturalCode, however, the entropy is measured in bits. Entropy in bits is simply $S = -\log_2(p)$, where p is the probability. The higher the entropy, therefore, the lower the probability and the more surprising a token or sequence of tokens is.

Based on the entropy equation, as probability approaches 0, entropy approaches infinity. An uncounted n-gram could exist, which would have 0 probability effectively canceling out all the other n-grams. Thus we rely on smoothing to address unseen n-grams.

UnnaturalCode uses Modified Kneser-Ney smoothing as implemented by `MITLM`, the MIT Language Model package [9]. Modified Kneser-Ney smoothing is widely regarded as a good choice for a general smoothing algorithm. This smoothing method discounts the probability of n-grams based on how many m-grams (where $m < n$) it must use to estimate their probability. Probability can be estimated for a 7-gram from a 3- and a 4-gram; in this case the probability will not be discounted as heavily as when probability must be estimated from seven 1-grams. Modified Kneser-Ney smoothing is tunable: a parameter may be set for each discount. In UnnaturalCode these parameters are not modified from their default values in `MITLM`.

While `MITLM` and the entropy estimation techniques implemented within `MITLM` were designed for natural-language text, UnnaturalCode employs these techniques on code. Hindle *et al.* [7] have shown that code has an even lower entropy per token than English text does per word, as shown in Figure 3. That is to say, the same techniques that work for natural English language texts work even better on source code. Moreover, syntactically invalid source code will often have a higher cross-entropy than compilable source code given a corpus of only syntactically valid source code. Therefore, defective source code looks *unnatural* to a natural language model trained on compilable source.

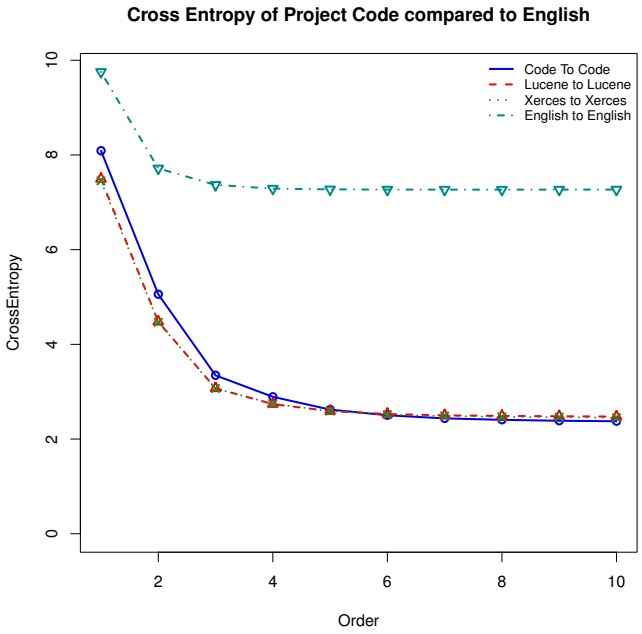

Figure 3: From Hindle *et al.* [7], a comparison of cross-entropy for English text and source code vs gram size, showing that English has much higher cross-entropy than code.

2.1 Previous Work

The frequency and importance of syntax errors among novice and experienced programmers has been studied by numerous authors [12, 11, 10, 17, 4, 16]. Many of these studies evaluated new programmers (often undergraduate first year students) and have examined the frequency of syntax errors and other kinds of errors. According to these studies missing semi-colons and misplaced braces cause between 10% to 20% of the errors that novices experience [12, 11, 10]. Jadud *et al.* [12, 11] took over 42000 snapshots of first year undergraduate code before each compile and labelled the cause of compilation failure. They found that 60% of syntax errors were immediately solved within 20 seconds, but 40% took longer, 5% taking longer than 10 minutes. Tabanao *et al.* [17, 18] studied student performance correlated with syntax errors, for more than 120 students, and found that the number of errors and frequency of compilation negatively correlated with student midterm grades. Jackson *et al.* [10] observed 559000 errors produced by 583 students during 1 semester; the second most common error was missing semicolons. Finally Kummerfeld *et al.* [14] studied the effect of student programming experience on syntax errors and found that experienced users relied on strategies to solve syntax errors; when these strategies failed, the experience programmers made erratic modifications just like their inexperienced counterparts. Thus we can see from numerous sources that syntax errors are a common source of errors that novice and experienced programmers run into and resolving these errors can often consume a lot of time and effort.

Previous publications that attempt to improve syntax error messages and syntax error location fall into two categories: *parser-based* or *type-based*. Burke's parse action deferral [2] is a *parser-based* technique that backs the parser down the parse stack when an error is encountered and then discards problematic tokens. Graham *et al.* [5] implemented a system that combined a number of heuristic and cost-based approaches including prioritizing which production rules will be resumed. Many modifications for particular parser algorithms have also been proposed to attempt to suppress spurious parse errors by repairing or resuming the parse after an error. Recent examples can be found by Kim *et al.* [13] who apply the k-nearest neighbour algorithm to search for repairs, or Corchuelo *et al.* [3] who present a modification that can be applied to parser generators and does not require user interaction. Other researchers have focused on *type-based* static analysis such as Heeren's Ph.D. thesis [6] which suggests implementing a constraint-based framework inside the compiler. Lerner *et al.* [15] use a corpus of compilable software to improve type error messages for statically typed languages. There have also been heuristic analyzers that work alongside the parser such as the one presented in

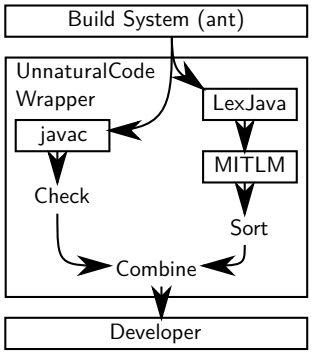

Figure 4: Data Flow of the Syntax-Error Location Detector, UnnaturalCode.

Hristova *et al.* [8], but this approach is limited to predefined heuristic rules addressing a specific selection of common mistakes. In comparison, UnnaturalCode requires no additional information about a language other than its lexemes, which may be extracted from its implementation or specification. The implementation presented here differs from those works in that it does not attempt to parse the source code. There is also a more recent publication by Weimer *et al.* [20] that uses Genetic Algorithms to mutate parse trees in an attempt to fix defects, but this requires syntactically valid source code.

3. A PROTOTYPE IMPLEMENTATION OF UNNATURALCODE

UnnaturalCode is designed to assist the programmer by locating the coding mistakes that caused a failed compilation. To this end, it only provides suggestions if the compile fails. If the compile succeeds it adds the error-free code to its corpus automatically. This allows UnnaturalCode to adapt rapidly to changing codebases. The data-flow diagram of UnnaturalCode is depicted in Figure 4.

The central component of the system is `javac-wrap.pl`, a wrapper script that is inserted in between the build system and the compiler, `javac`. Our tests were run with the `ant` build system and the Oracle 1.7.1 JDK `javac`.

In the case of a failed compilation, UnnaturalCode lexically analyzes the source file which failed to compile, and queries the smoothed n-gram model using a sliding window of length $2n$ tokens. Both the query code and the corpus has comments removed and contiguous whitespace reduced to a single space.

Once UnnaturalCode has finished computing results for all the queries, it ranks them by the entropy per token of each query, as in Figure 5. It then reports the top five strings with the highest entropy to the user as suggestions of where to look for mistakes in the code. The entropy is a measure of how unlikely the presented string of $2n$ tokens is, given the corpus. In this case, $n = 10$, a setting that works well while also keeping memory use low enough for machines with 512MB of memory.

The entropy, S, is calculated by `MITLM` in bits as the negative logarithm of the probability. The higher the entropy score, the less likely a given string of tokens was found to be by `MITLM`. Figure 3 shows that entropy per token values are typically between 2 and 3 bits, compared to English text which typically has entropy near 7 bits per word.

Check near == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return
With entropy 4.552985
Check near () == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS
With entropy 4.498802
Check near NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return;
With entropy 4.244520
Check near) lastDoc = NO_MORE_DOCS; return; }
With entropy 4.183379
Check near) == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS;
With entropy 3.858807

Figure 5: Example of UnnaturalCode output showing an accurate error location from the motivational example.

Table 1: Validation Data Summary Statistics

n-gram order	10
Files in Lucene 4.0.0	2 866
Files in Lucene 4.1.0	2 916
Files in Ant 1.7.0	1 113
Files in Ant 1.8.4	1 196
Files in Xerces Java 2.9.1	716
Files in Xerces Java 2.10.0	754
Files in Xerces Java 2.11.0	757
Types of mutation	3
Mutations per type per file/hunk	≥ 120
Tests using Lucene 4.0.0 corpus	2 626 940
Tests using Ant 1.7.0 corpus	833 960
Tests using XercesJ 2.9.1 corpus	1 547 180
Total Tests Performed	5 008 080

The current implementation is fast enough to be used interactively by a software engineer. Building the corpus takes much less time than compiling the same code, because only lexical analysis is performed. For the Lucene 4.0.0 corpus, results for a broken compile, in the form of suggestions, are found for a source file with over 1000 tokens in under 0.02 seconds on an Intel i7-3770 running Ubuntu 12.10 using only a single core and under 400MiB of memory. This is more than fast enough to be used interactively if `MITLM` is already running. However, `MITLM` start-up can be quite slow depending on the size of the corpus. For the Lucene 4.0.0 corpus `MITLM` takes about 5 seconds to start on the same platform.

Since UnnaturalCode is not a parser, it is not necessary for the model to recieve an unmatched `}` to detect a missing `{`. UnnaturalCode instead relies on nearby contextual information such as `public static void main(String[] args)`. Therefore, the length of a body, block, parethentical expression or other balanced syntactic structure is irrelevant despite the limited 20-token sliding window that UnnaturalCode operates with.

4. VALIDATION METHOD

UnnaturalCode was tested primarily on three Apache Foundation projects: Lucene,[4] Ant,[5] and XercesJ.[6]

For each experiment the training corpus consisted of every Java source file from the oldest version of a single project.

[4]Apache Lucene is available at `https://lucene.apache.org/`
[5]Apache Ant is available at `https://ant.apache.org/`
[6]Apache XercesJ is available at `https://xerces.apache.org/xerces-j/`

These source files were compiled and all compiled files were added to the training corpus. This corpus was used for the duration of testing. No automatic updates to the training corpus were performed during testing.

The following mutations were applied to files:

- **Random Deletion:** a token (lexeme) was chosen at random from the input source file and deleted. The file was then run through the querying and ranking process to determine where the first result with adjacent code appeared in the suggestions.
- **Random Replacement:** a token was chosen at random and replaced with a random token found in the same file.
- **Random Insertion:** a location in the source file was chosen at random and a random token found in the same file was inserted there.

The resulting mutant files were actually compiled, and when compilation succeeded the mutant file was skipped. This had particularly dramatic results on the deletion tests, where 33% of the random token deletions resulted in a file that still compiled.

After compilation, the compiler's own error messages were considered. These error messages were also given a score for each file. The compiler was scored in a similar fashion to UnnaturalCode: the first result produced by the compiler mentioning the correct line number was considered correct.

Each of these three tests was repeated on each input file at least 120 times, each time modifying a newly and randomly chosen location in the source file. For Lucene, all 3 tests were performed at least 120 times each on 1266 files. Millions of tests were run on Ant and XerecesJ as well. Thus, a total of over 5 million data points were collected as shown in Table 1.

For Lucene, 4 different kinds of source-code inputs were tested. First, for the Lucene 4.0.0 test, source files were taken from the exact same package as the corpus and were modified by the above process and then tested. These source files exist unmodified in the corpus. Second, source files were taken from the next Lucene release, the 4.1.0 version, that had been modified by developers. Some of these source files exist in their 4.0.0 form in the corpus, but have been modified by developers and then by the above process. These files are listed in the results as the "Lucene 4.1.0 – Changed Files" test. Additionally, new source files were added to Lucene after the 4.0.0 release for 4.1.0. These new files do not exist in the corpus but are related to files which did. These are listed in the results as the "Lucene 4.1.0 – New Files" test. Finally, to test files completely external to the corpus, Java source files from Apache Ant 1.8.4 were tested. Not only do these files not exist in the corpus but they are not related to the files that do, except in that they are both Apache Foundation software packages.

In order to get the above results, the following steps were performed. First a corpus was created from the earliest release. For example, Lucene 4.0.0 was built, automatically adding all compilable source files to the corpus.

Next, we ran query tests. In each test, we choose a random token in the input file to mutate as described above. Then, we run UnnaturalCode on the input file and record the rank of the first correct result, r_q.

The rankings are analyzed statistically using the reciprocal rank. The mean is reported as the mean reciprocal rank (MRR) [19]:

$$\mu = \frac{1}{|\mathcal{Q}|}\left[\sum_{q\in\mathcal{Q}}\frac{1}{r_q}\right].$$

$\mathcal{Q}$ is the set of all queries, and q is an individual query from that set. For example, $|\mathcal{Q}| = 120$ for an individual file and type of mutation. Using the MRR has several advantages: it differentiates the most among the first few (highest) ranks, and has a worst value of 0, whereas for UnnaturalCode the worst possible absolute rank depends on the length of the input file. This is important because the more results a programmer has to consider to find a syntax error, the less likely he or she is to consider them all.

MRR is a very unforgiving measure of the performance of a system that returns multiple sorted results. In order to achieve an MRR greater than 0.75, the correct result must be the first result presented to the user most of the time. For example, consider three hypothetical token deletions performed as described above on a single file. If the correct result was ranked first for the first test, second for the second test, and third for the third test, UnnaturalCode would only have achieved an MRR score of 0.61 for that file.

MRR scoring was implemented for two different sets of interleaved JavaC and UnnaturalCode results. These combined results consist of a JavaC result followed by an UnnaturalCode result, followed by a JavaC result, and so on. The two variations are: 1) returning a JavaC result first; and 2) returning an UnnaturalCode result first. These combined results represent the intended use of UnnaturalCode as a way to augment compiler errors.

5. VALIDATION RESULTS

Figure 6 shows the distributions of the MRR scores of the files of versions of Lucene and Ant versus a Lucene trained corpus. The wider the shaded area is in these charts, the more files had that MRR. These plots also show the 25th, 50th and 75th percentiles as the beginning of the black box, the white dot, and the end of the black box in the centre. Table 2 presents the cumulative mean MRRs for each data set and method.

UnnaturalCode performs very well at detecting mutations in code that it is familiar with. UnnaturalCode did very well with only the first Lucene 4.0.0 version in the corpus when tested against both Lucene 4.0.0 and Lucene 4.1.0. A test of Ant 1.8.4 against a foreign corpus (Lucene 4.0.0) results in poor performance. Syntax error detection performance is best with a corpus trained on the same or earlier version of the system.

The scores and chart for the whole of Lucene 4.1.0 is not the entire story for that version. It contains three kinds of files: files unchanged from 4.0.0, changed files, and new files added since 4.0.0. Figure 7 clearly shows how these types files bring the MRR scores for 4.1.0 down from the scores for 4.0.0. The newly added files have very inconsistent performance with MRR scores near those of the scores for Ant's unrelated files, despite the fact that they are a part of the same project as the training corpus. Figures 8 and 9 show the same pattern on Apache Ant and XercesJ.

Figure 10 compares the performance of Ant 1.7.0 versus itself and Ant 1.8.4. The MRR behaviour is similar to the

Table 2: Cumulative Mean Reciprocal Ranks (Mean MRR)

Sources Tested	Corpus	Delete	Insert	Replace
Lucene 4.0.0	Lucene 4.0.0	0.88	0.99	0.98
Lucene 4.1.0	Lucene 4.0.0	0.77	0.91	0.91
Ant 1.8.4	Lucene 4.0.0	0.20	0.36	0.36
Lucene 4.1.0 Only new files	Lucene 4.0.0	0.30	0.47	0.48
Lucene 4.1.0 Only changed files	Lucene 4.0.0	0.68	0.86	0.85
Ant 1.7.0	Ant 1.7.0	0.86	0.99	0.98
Ant 1.8.4	Ant 1.7.0	0.55	0.75	0.74
Ant 1.8.4 Only new files	Ant 1.7.0	0.29	0.54	0.53
Ant 1.8.4 Only changed files	Ant 1.7.0	0.42	0.66	0.66
XercesJ 2.9.1	XercesJ 2.9.1	0.86	0.98	0.98
XercesJ 2.10.0	XercesJ 2.9.1	0.50	0.80	0.79
XercesJ 2.11.0	XercesJ 2.9.1	0.49	0.79	0.78
XercesJ 2.11.0 Only new files	XercesJ 2.9.1	0.25	0.46	0.47
XercesJ 2.11.0 Only changed files	XercesJ 2.9.1	0.50	0.81	0.79

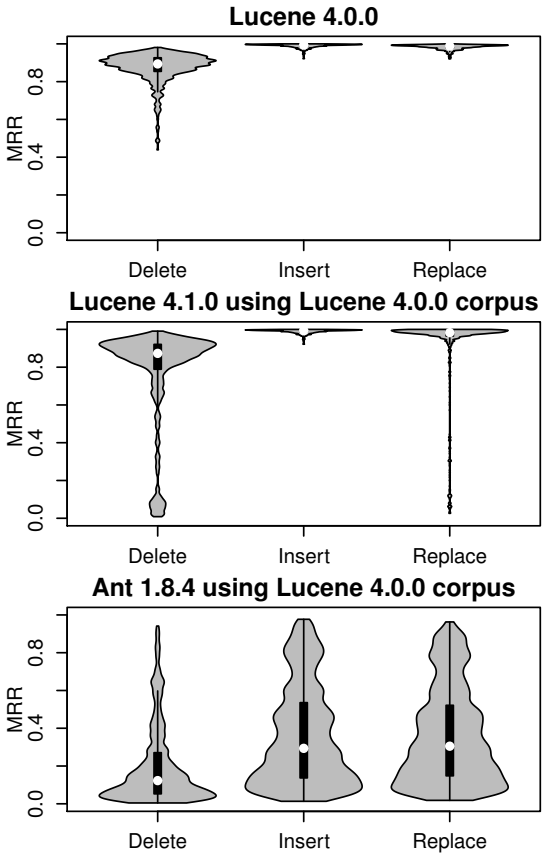

Figure 6: UnnaturalCode-only MRR Distributions of the files of Lucene 4.0.0, 4.1.0 and the files of Ant 1.8.4 tested against a Lucene 4.0.0 corpus.

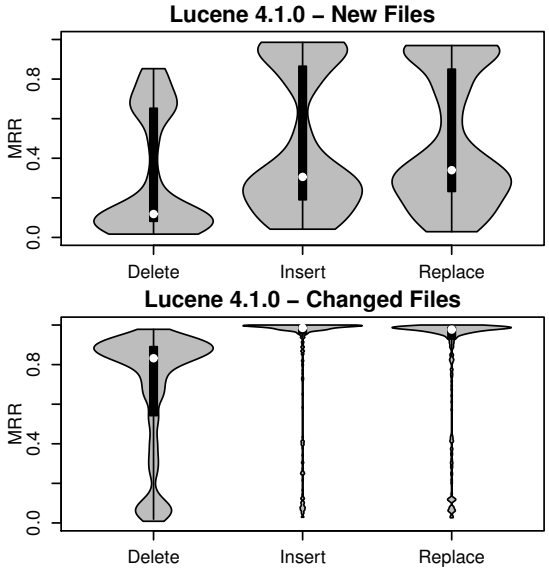

Figure 7: UnnaturalCode-only MRR Distributions of only new and changed files from Lucene 4.1.0, using Lucene 4.0.0 as the corpus.

Lucene plots in Figure 6 for Lucene versus Lucene tests, and the poor performance of Ant versus Lucene has been negated by using Ant code in the corpus. The median remains very high, implying that UnnaturalCode scores very well on most files.

Figure 11 tests 3 consecutive major releases of XercesJ against a corpus of XercesJ 2.9.1. In all cases the median MRR of these tests are above or near 0.5, which means that over 50% of the files have an MRR greater to or near 0.5. This is the same MRR score that a hypothetical system which always returns the correct result in second place

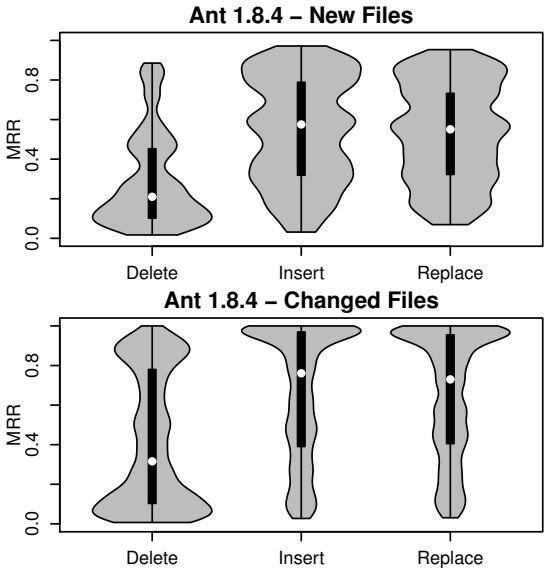

Figure 8: UnnaturalCode-only MRR Distributions of only new and changed files from Apache Ant 1.8.4, using ant 1.7.0 as the corpus.

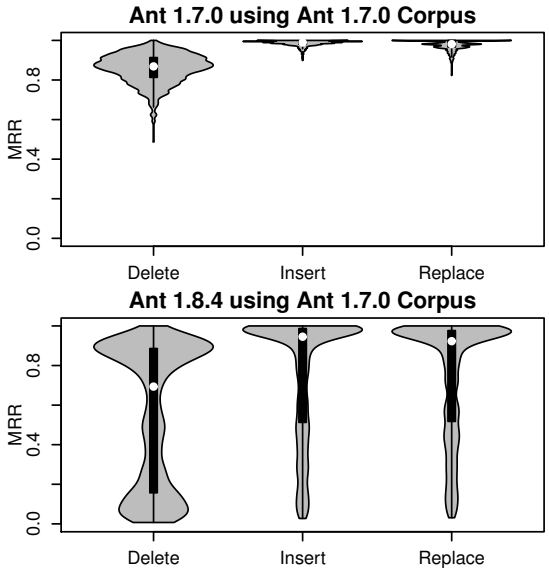

Figure 10: UnnaturalCode-only MRR Distributions of the files of Ant 1.7.0 and Ant 1.8.4 tested against the Ant 1.7.0 corpus.

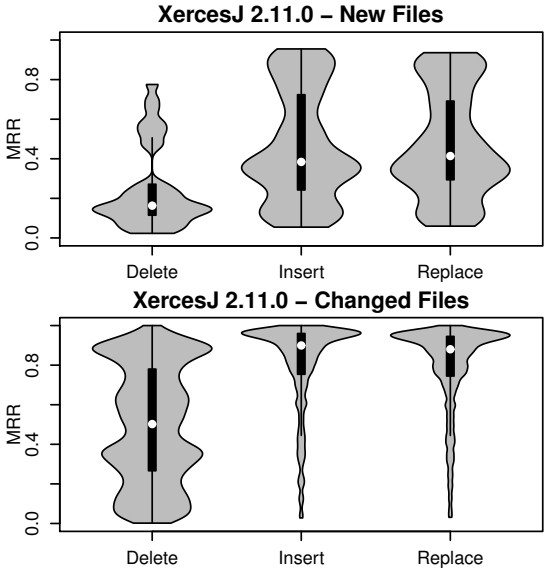

Figure 9: UnnaturalCode-only MRR Distributions of only new and changed files from XercesJ 2.11.0 Using XercesJ 2.9.1 as the corpus.

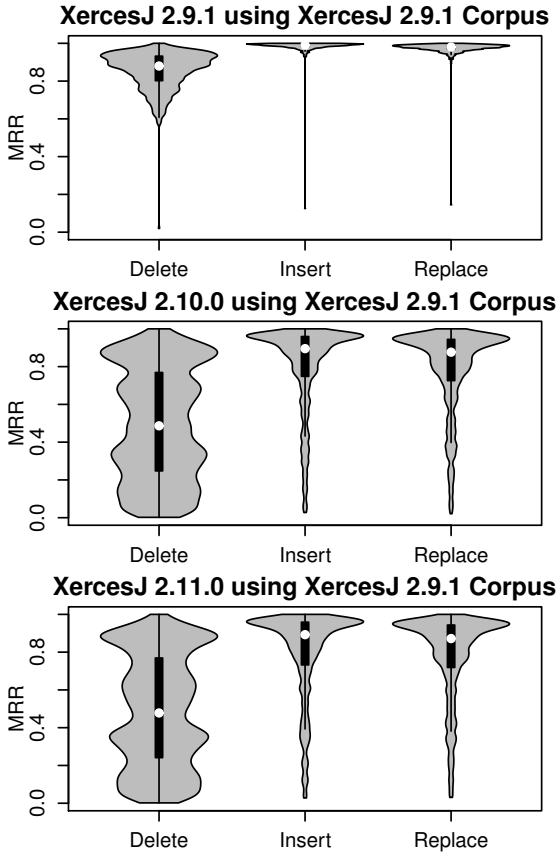

Figure 11: MRR Distributions of the files of XercesJ 2.9.1, 2.10.0, and 2.11.0 tested against the XercesJ 2.9.1 corpus.

would get. As with Ant, for XercesJ in the test on a past corpus (XercesJ 2.10.0 and 2.11.0) the bottom quartile extends further, but ends before an MRR score of 0.2, which implies that for 75% of the files tested, UnnaturalCode performed similarly to, or better than, a hypothetical system which always returns the correct result in fifth place.

Table 3 shows the results of the compiler integration tests and Figure 12 shows the MRR distribution for one of these tests. In this table, the column heading describes the output interleaving pattern: "UUUU" gives MRR means for UnnaturalCode results, "JJJJ" gives MRR means for JavaC, "JUJU"

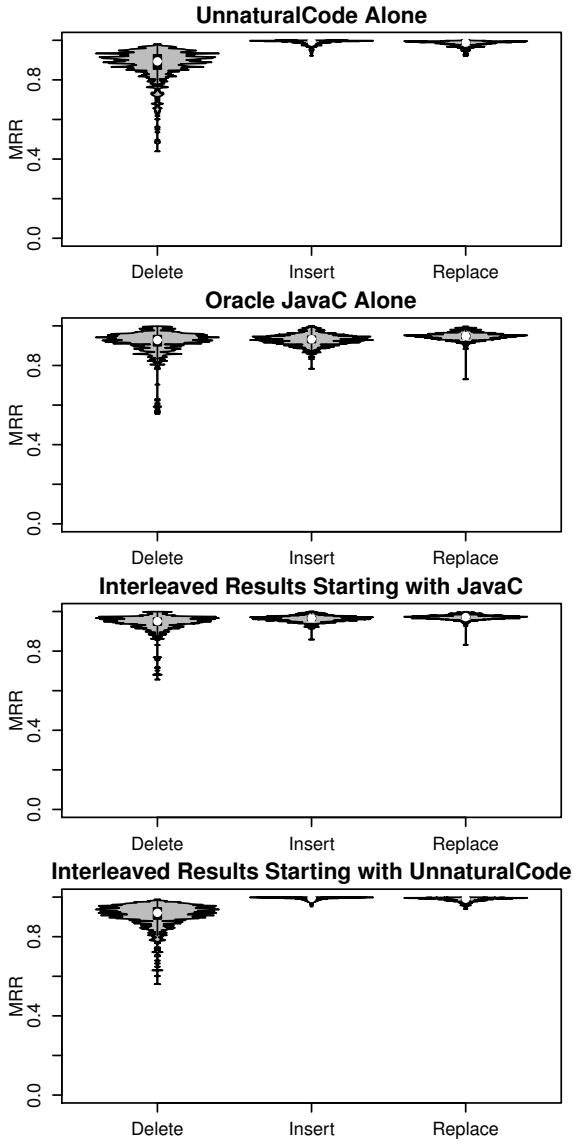

Figure 12: MRR Distributions of the files of Lucene 4.0.0 for the compiler integration test.

gives MRR means for interleaved results with JavaC's first result first in the output, and "UJUJ" gives MRR means for interleaved results starting with UnnaturalCode's first result. All four plots come from the same set of randomly chosen mutations. Both JavaC and UnnaturalCode perform well on their own. UnnaturalCode performs worse than JavaC on the deletion test and better than JavaC on the insertion and replacement tests. However, interleaved results perform better than either system by itself: the best performing interleave depends on the file.

6. DISCUSSION

The n-gram language-model approach was capable of detecting all mutations: inserted tokens, missing tokens, and replaced tokens. This is because the sequence of tokens will not have been seen before by the language model, assuming it has been trained on compilable code.

6.1 Performance on Milestones

In the milestone tests, some files would get consistently wrong results because the top results would always be the same regardless of where the changes were made. In particular, files from outside the corpus that contained strings of new identifier tokens would consistently produce poor results with UnnaturalCode.

Unfortunately, with only a single project in the corpus, performance was sometimes very poor. This poor performance could be easily triggered by adding new identifiers that were not present in the corpus, since those new identifiers were labelled with high entropy by the model. Sometimes this behaviour is accurate, as in the case of a misspelled identifier, but sometimes it is inaccurate, as in the case of a newly added, but correctly spelled, identifier.

These results are highly encouraging, however. Even the worst MRR score of 0.20 implies that there are files in Apache Ant in which errors are locatable using a corpus trained on Lucene. If one runs the tests as described above on broken Java code that imports classes from packages that do not exist in the training corpus, these correctly specified imports will almost always be the top five results. However, this behaviour only persists until they are added to the corpus, and this happens after the first successful compilation of those files. For example, even though class imports in Ant 1.8.4 share the same `org.apache` class path, they also contain new identifiers. When `ProjectHelperImpl.java` imports `org.apache.tools.ant.Target`, this string contains three identifiers, "tools," "ant," and "Target" which are not in the training corpus. This will cause the average entropy of any string of 20 tokens containing "`tools.ant.Target`" to contain at least three high entropy contributions. In comparison, when the code is mutated for testing, at most one additional unseen token is introduced. One possible solution to the new identifier problem is to ignore identifier content and train on the type of the identifier lexeme itself [1]. The syntax error detector can often misreport new identifiers, such as package names, as syntax errors.

6.2 Performance with Compiler Integration

The UnnaturalCode/JavaC combination performed better than either JavaC or UnnaturalCode alone in every test case.

UnnaturalCode was often capable of detecting errors that JavaC was not and JavaC often was capable of detecting errors that UnnaturalCode was not. This fact allows UnnaturalCode to augment the accuracy, in terms of MRR score, of the compiler when its results were interleaved. The interleaving improved the MRR score from 0.915, using just JavaC, to 0.943 using interleaved results for the deletion test on Lucene 4.0.0. Intuitively, this means that 66% of the time when JavaC's highly ranked results are wrong, UnnaturalCode has a highly ranked result that is correct. Or, we can save 66% of the time that software engineers spend hunting for the source of syntax errors. Results are presented to the user side-by-side because the best interleave choice is not consistent.

7. THREATS TO VALIDITY

Construct validity is affected by the assumption that single-token mutation is representative of syntax errors. This as-

Table 3: Compiler Integration Mean Reciprocal Ranks (MRRs).

		Interleaving Pattern			
Sources Tested	Corpus	UUUU	JJJJ	JUJU	UJUJ
Lucene 4.0.0	Lucene 4.0.0	.950	.932	.959	.963
Lucene 4.1.0	Lucene 4.0.0	.865	.937	.960	.914
Ant 1.7.0	Ant 1.7.0	.940	.927	.956	.958
Ant 1.8.4	Ant 1.7.0	.681	.923	.945	.806
Ant 1.8.4	Lucene 4.0.0	.308	.921	.930	.600
XercesJ 2.9.1	XercesJ 2.9.1	.939	.895	.937	.955
XercesJ 2.10.0	XercesJ 2.9.1	.694	.889	.916	.796
XercesJ 2.11.0	XercesJ 2.9.1	.688	.884	.911	.791

sumption may not be very representative of changes a software engineer would make between compilation attempts.

> However, single-token mutations are the worst-case scenario and provide a lower bound on performance.

A file with two syntax errors is further from compiling than a file with one syntax error. Since UnnaturalCode only models files that do compile, a file with two syntax errors will have more entropy than a file with one syntax error. In this case a developer would be presented with the highest-entropy error first. If they chose to fix that problem and recompile, they would be presented with the next-highest entropy error. Multi-token errors would be easier for UnnaturalCode to detect.

Internal validity is hampered by using the MRR formula to score our ranking of the correct query results. These rankings are affected by the maximum gram length, n, where typically larger is better, and the number of total results. The number of total results is $l - 2n/s$ where l is the the length of the file and s is the step size. Since correct results comprise at most $2n/s$ of the results, the chance of the correct result appearing in the top 5 if the results are sorted randomly is approximately proportional to $1/l$. In other words, the system will naturally perform better on short input files simply because there is less noise to consider.

External validity is affected by the choice of Java projects. The experimental evaluation covers 5 million tests of UnnaturalCode and JavaC across 3 different medium-to-large Apache Foundation Java projects: Ant, XercesJ and Lucene. JavaC compilation is much slower than querying UnnaturalCode alone.

Java has a syntax typical of many C-like languages. Thus, these tests are a good representative, and the results for these projects will generalize well to other projects. These results should generalize to other languages as well but this assumption has not been tested yet. The evaluation on 5 million single-token mutation tests across 3 distinct corpuses is a fairly significant evaluation of the new method.

8. FUTURE DIRECTIONS

This technology can be very useful to software engineers who are actively developing a piece of software. All the engineer must do is instruct their build system to call the wrapper instead of the compiler directly, and the system will begin building a corpus; if the compile fails it will return side-by-side ranked results from both the compiler and from UnnaturalCode. The performance evaluation indicates that a system using the ideas presented in this paper will suggest the location of the fault in the top two results often enough to be useful as long as the corpus contains one successful compile of the same project.

The method proposed should be implemented not only as a compiler wrapper for general purpose command-line use, but also as a plug-in for an integrated development environment (IDE) such as Eclipse. In such an environment it could provide immediate and visual feedback after a failed compile of which lines were likely to cause problems, perhaps by colouring the background with a colour corresponding to the increase in entropy for those lines.

The generalization to other languages needs to be evaluated. One simply needs to replace the Java lexical analyzer used in UnnaturalCode with a lexical analyzer for the language they wish to use and modify the compiler wrapper to locate input file arguments and detect the failure of their compiler of choice. The n-gram model is flexible and will work with many programming languages [7].

Several changes could be made to enhance performance. The first is to build an n-gram model consisting only of the lexical types of each token, but not the token itself. The language model corpus and input would then consist of words like "identifier," "operator," "type," instead of "readByte", "+," and "int." Combining entropy estimates from that model with the current model could produce more accurate results when considering never-before-seen tokens. This approach would be similar to the model proposed in Allamanis *et al.* [1].

The effect of a multi-project corpus on syntax error detection should be investigated. The idea is to explore what makes a good corpus for general purpose syntax error detection.

Several extensions to the system could be implemented. For example, the combination of this method with token prediction could automatically repair source code by statistically directing most-likely syntax repair searches based on a dynamic and project-specific corpus instead of statically defined least-cost repair searches such as those presented in Corchuelo *et al.* [3] and Kim *et al.* [13]. This approach should be much more efficient than the approach applied in Weimer *et al.* [20]. It may also be interesting for developers or project managers to be able to see the entropy of each line or token of source code themselves because the entropy may correlate with other properties of the source code that have not yet been considered.

9. CONCLUSION

This paper presents a system to address the common issue of poor syntax-error location reporting by modern compilers.

Syntax errors plague novice and experienced programmers alike. UnnaturalCode avoids parsing completely, instead relying on a statistical language model from the field of natural language processing. The choice of model was guided by the promising results in code completion. UnnaturalCode consists of a compiler wrapper, a lexical analyzer, and a language model that cooperate and are more than fast enough to be used interactively. UnnaturalCode is tested by inserting, deleting and replacing random tokens in files from the software that it was trained on, and later versions of the software it was trained on, and a completely independent software package. The experimental evaluation shows that UnnaturalCode performed very well on the same version, well on the next version, and poorly on external software. When UnnaturalCode's results are combined with the compilers own results, the correct location is reported earlier than with the compiler's results alone.

This work helps bridge the gap between statistical language analysis tools and computer-language parsers. Bridging this gap solves a problem that has haunted the users of parsers since they were invented: accurately locating syntax errors. This approach is unique in that it employs a statistical model: all previous work in this area are based on parser modifications to repair or recover from syntax errors, static code analysis, predefined heuristics, or type analysis. Furthermore, the model can evolve with the changing contexts of individual engineers and projects as it continuously updates its training corpus. By combining this method with the current compiler's error messages we achieve better performance in syntax error localization than either technique alone.

The n-gram language-model is capable of enhancing the compiler's ability to locate missing tokens, extra tokens, and replaced tokens.

Finally, when you build a language model of working source code, "syntax errors just aren't natural."

10. ACKNOWLEDGMENTS

Abram Hindle is supported by an NSERC Discovery Grant.

11. REFERENCES

[1] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[2] M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(2):164–197, Mar. 1987.

[3] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, Nov. 2002.

[4] S. Garner, P. Haden, and A. Robins. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pages 173–180. Australian Computer Society, Inc., 2005.

[5] S. L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. *SIGPLAN Not.*, 14(8):168–175, Aug. 1979.

[6] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, Nederlands, 2005.

[7] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847, June 2012.

[8] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[9] B. Hsu and J. Glass. Iterative language model estimation: efficient data structure & algorithms. 2008.

[10] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*, pages T4C–T4C. IEEE, 2005.

[11] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[12] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[13] I.-S. Kim and K.-M. Choe. Error repair with validation in LR-based parsing. *ACM Trans. Program. Lang. Syst.*, 23(4):451–471, July 2001.

[14] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 105–111. Australian Computer Society, Inc., 2003.

[15] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 425–434, San Diego, CA, USA, 2007.

[16] L. McIver. The effect of programming language on error rates of novice programmers. In *12th Annual Workshop of the Psychology of Programming Interest Group*, pages 181–192. Citeseer, 2000.

[17] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Identifying at-risk novice java programmers through the analysis of online protocols. In *Philippine Computing Science Congress*, 2008.

[18] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 85–92, New York, NY, USA, 2011. ACM.

[19] E. M. Voorhees et al. The TREC-8 question answering track report. In *Proceedings of TREC*, volume 8, pages 77–82, 1999.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.