Software testing and the naturally occurring data assumption in natural language processing*

K. Bretonnel Cohen **William A. Baumgartner, Jr.** **Lawrence Hunter**

Abstract

It is a widely accepted belief in natural language processing research that naturally occurring data is the best (and perhaps the only appropriate) data for testing text mining systems. This paper compares code coverage using a suite of functional tests and using a large corpus and finds that higher class, line, and branch coverage is achieved with structured tests than with even a very large corpus.

1 Introduction

In 2006, Geoffrey Chang was a star of the protein crystallography world. That year, a crucial component of his code was discovered to have a simple error with large consequences for his research. The nature of the bug was to change the signs (positive versus negative) of two columns of the output. The effect of this was to reverse the predicted "handedness" of the structure of the molecule—an important feature in predicting its interactions with other molecules. The protein for his work on which Chang was best known is an important one in predicting things like human response to anticancer drugs and the likelihood of bacteria developing antibiotic resistance, so his work was quite influential and heavily cited. The consequences for Chang were the withdrawal of 5 papers in some of the most prestigious journals in the world. The consequences for the rest of the scientific community have not been quantified, but were substantial: prior to the retractions, publishing papers with results that did not jibe with his model's predictions was difficult, and obtaining grants based on preliminary results that seemed to contradict his published results was difficult as well. The Chang story (for a succinct discussion, see (Miller, 2006), and see (Chang et al., 2006) for the retractions) is an object illustration of the truth of Rob Knight's observation that "For scientific work, bugs don't just mean unhappy users who you'll never actually meet: they mean retracted publications and ended careers. It is critical that your code be fully tested before you draw conclusions from results it produces" (personal communication). Nonetheless, the subject of software testing has been largely neglected in academic natural language processing. This paper addresses one aspect of software testing: the monitoring of testing efforts via code coverage.

1.1 Code coverage

Code coverage is a numerical assessment of the amount of code that is executed during the running of a test suite (McConnell, 2004). Although it is by no means a completely sufficient method for determining the completeness of a testing effort, it is nonetheless a helpful member of any suite of metrics for assessing testing effort completeness. Code coverage is a metric in the range 0-1.0. A value of 0.86 indicates that 86% of the code was executed while running a given test suite. 100% coverage is difficult to achieve for any nontrivial application, but in general, high degrees of "uncovered" code should lead one to suspect that there is a large amount of

*K. Bretonnel Cohen is with The MITRE Corporation. All three co-authors are at the Center for Computational Pharmacology in the University of Colorado School of Medicine.

Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 23–30,
Columbus, Ohio, USA, June 2008.

code that might harbor undetected bugs simply due to never having been executed. A variety of code coverage metrics exist. *Line coverage* indicates the proportion of lines of code that have been executed. It is not the most revealing form of coverage assessment (Kaner et al., 1999, p. 43), but is a basic part of any coverage measurement assessment. *Branch coverage* indicates the proportion of branches within conditionals that have been traversed (Marick, 1997, p. 145). For example, for the conditional `if $a && $b`, there are two possible branches—one is traversed if the expression evaluates to `true`, and the other if it evaluates to `false`. It is more informative than line coverage. *Logic coverage* (also known as *multicondition coverage* (Myers, 1979) and *condition coverage* (Kaner et al., 1999, p. 44) indicates the proportion of sets of variable values that have been tried—a superset of the possible branches traversed. For example, for the conditional `if $a || $b`, the possible cases (assuming no short-circuit logic) are those of the standard (logical) truth table for that conditional. These coverage types are progressively more informative than line coverage. Other types of coverage are less informative than line coverage. For example, *function coverage* indicates the proportion of functions that are called. There is no guarantee that each line in a called function is executed, and all the more so no guarantee that branch or logic coverage is achieved within it, so this type of coverage is weaker than line coverage. With the advent of object-oriented programming, function coverage is sometimes replaced by *class coverage*—a measure of the number of classes that are covered.

We emphasize again that code coverage is not a sufficient metric for evaluating testing completeness in isolation—for example, it is by definition unable to detect "errors of omission," or bugs that consist of a failure to implement needed functionality. Nonetheless, it remains a useful part of a larger suite of metrics, and one study found that testing in the absence of concurrent assessment of code coverage typically results in only 50-60% of the code being executed ((McConnell, 2004, p. 526), citing Wiegers 2002).

We set out to question whether a dominant, if often unspoken, assumption of much work in contemporary NLP holds true: that feeding a program a large corpus serves to exercise it adequately. We began with an information extraction application that had been built for us by a series of contractors, with the contractors receiving constant remote oversight and guidance but without ongoing monitoring of the actual code-writing. The application had benefitted from no testing other than that done by the developers. We used a sort of "translucent-box" or "gray-box" paradigm, meaning in this case that we treated the program under test essentially as a black box whose internals were inaccessible to us, but with the exception that we inserted hooks to a coverage tool. We then monitored three types of coverage—line coverage, branch coverage, and class coverage—under a variety of contrasting conditions:

- A set of developer-written functional tests versus a large corpus with a set of semantic rules optimized for that corpus.
- Varying the size of the rule set.
- Varying the size of the corpus.

We then looked for coverage differences between the various combinations of input data and rule sets. In this case, the null hypothesis is that no differences would be observed. In contrast with the null hypothesis, the unspoken assumption in much NLP work is that the null hypothesis does not hold, that the primary determinant of coverage will be the size of the corpus, and that the observed pattern will be that coverage is higher with the large corpus than when the input is not a large corpus.

2 Methods and materials

2.1 The application under test

The application under test was an information extraction application known as OpenDMAP. It is described in detail in (Hunter et al., 2008). It achieved the highest performance on one measure of the protein-protein interaction task in the BioCreative II shared task (Krallinger et al., 2007). Its use in that task is described specifically in (Baumgartner Jr. et al., In press). It contains 7,024 lines of code spread across three packages (see Table 1). One major package deals with representing the semantic grammar rules themselves, while the other deals with applying the rules to and extracting data from arbitrary textual input. (A minor package deals with

Component	Lines of code
Parser	3,982
Rule-handling	2,311
Configuration	731
Total	7,024

Table 1: Distribution of lines of code in the application.

the configuration files and is mostly not discussed in this paper.)

The rules and patterns that the system uses are typical "semantic grammar" rules in that they allow the free mixing of literals and non-terminals, with the non-terminals typically representing domain-specific types such as "protein interaction verb." Non-terminals are represented as classes. Those classes are defined in a Protégé ontology. Rules typically contain at least one element known as a *slot*. Slot-fillers can be constrained by classes in the ontology. Input that matches a slot is extracted as one of the participants in a relation. A limited regular expression language can operate over classes, literals, or slots. The following is a representative rule. Square brackets indicate slots, curly braces indicate a class, the question-mark is a regular expression operator, and any other text is a literal.

```
  {c-interact} := [interactor1]
{c-interact-verb} the?
[interactor2]
```

The input *NEF binds PACS-2* (PMID 18296443) would match that rule. The result would be the recognition of a protein interaction event, with interactor1 being *NEF* and interactor2 being *PACS-2*. Not all rules utilize all possibilities of the rule language, and we took this into account in one of our experiments; we discuss the rules further later in the paper in the context of that experiment.

2.2 Materials

In this work, we made use of the following sets of materials.

- A large data set distributed as training data for part of the BioCreative II shared task. It is described in detail in (Krallinger et al., 2007). Briefly, its domain is molecular biology, and in particular protein-protein interactions—an important topic of research in computational bioscience, with significance to a wide range of topics in biology, including understanding the mechanisms of human diseases (Kann et al., 2006). The corpus contained 3,947,200 words, making it almost an order of magnitude larger than the most commonly used biomedical corpus (GENIA, at about 433K words). This data set is publicly available via `biocreative.sourceforge.net`.

- In conjunction with that data set: a set of 98 rules written in a data-driven fashion by manually examining the BioCreative II data described just above. These rules were used in the BioCreative II shared task, where they achieved the highest score in one category. The set of rules is available on our SourceForge site at `bionlp.sourceforge.net`.

- A set of functional tests created by the primary developer of the system. Table 2 describes the breakdown of the functional tests across various aspects of the design and functionality of the application.

Test type	Number of tests
Basic	85
Pattern/rule	67
Patterns only	90
Slots	9
Slot nesting	7
Slot property	20
Total	278

Table 2: Distribution of functional tests.

2.3 Assessing coverage

We used the open-source Cobertura tool (Mark Doliner, personal communication; `cobertura.sourceforge.net`) to measure coverage. Cobertura reports line coverage and branch coverage on a per-package basis and, within each package, on a per-class basis[1].

The architecture of the application is such that Cobertura's per-package approach resulted in three

[1] Cobertura is Java-specific. PyDEV provides code coverage analysis for Python, as does Coverage.py.

sets of coverage reports: for the configuration file processing code, for the rule-processing code, and for the parser code. We report results for the application as a whole, for the parser code, and for the rule-processing code. We did note differences in the configuration code coverage for the various conditions, but it does not change the overall conclusions of the paper and is omitted from most of the discussion due to considerations of space and of general interest.

3 Results

We conducted three separate experiments.

3.1 The most basic experiment: test suite versus corpus

In the most basic experiment, we contrasted class, line, and branch coverage when running the developer-constructed test suite and when running the corpus and the corpus-based rules. Tables 3 and 4 show the resulting data. As the first two lines of Table 3 show, for the entire application (parser, rule-handling, and configuration), line coverage was higher with the test suite—56% versus 41%—and branch coverage was higher as well—41% versus 28% (see the first two lines of Table 3).

We give here a more detailed discussion of the results for the entire code base. (Detailed discussions for the parser and rule packages, including granular assessments of class coverage, follow.)

For the parser package:

- Class coverage was higher with the test suite than with the corpus—88% (22/25) versus 80% (20/25).
- For the entire parser package, line coverage was higher with the test suite than with the corpus—55% versus 41%.
- For the entire parser package, branch coverage was higher with the test suite than with the corpus—57% versus 29%.

Table 4 gives class-level data for the two main packages. For the parser package:

- Within the 25 individual classes of the parser package, line coverage was equal or greater with the test suite for 21/25 classes; it was not just equal but greater for 14/25 classes.
- Within those 21 of the 25 individual classes that had branching logic, branch coverage was equal or greater with the test suite for 19/21 classes, and not just equal but greater for 18/21 classes.

For the rule-handling package:

- Class coverage was higher with the test suite than with the corpus—100% (20/20) versus 90% (18/20).
- For the entire rules package, line coverage was higher with the test suite than with the corpus—63% versus 42%.
- For the entire rules package, branch coverage was higher with the test suite than with the corpus—71% versus 24%.

Table 4 gives the class-level data for the rules package:

- Within the 20 individual classes of the rules package, line coverage was equal or greater with the test suite for 19/20 classes, and not just equal but greater for 6/20 classes.
- Within those 11 of the 20 individual classes that had branching logic, branch coverage was equal or greater with the test suite for all 11/11 classes, and not just equal but greater for (again) all 11/11 classes.

3.2 The second experiment: Varying the size of the rule set

Pilot studies suggested (as later experiments verified) that the size of the input corpus had a negligible effect on coverage. This suggested that it would be worthwhile to assess the effect of the rule set on coverage independently. We used simple ablation (deletion of portions of the rule set) to vary the size of the rule set.

We created two versions of the original rule set. We focussed only on the non-lexical, relational pattern rules, since they are completely dependent on the lexical rules. Each version was about half the

Metric	Functional tests	Corpus, all rules	nominal rules	verbal rules
Overall line coverage	**56%**	41%	41%	41%
Overall branch coverage	**41%**	28%	28%	28%
Parser line coverage	**55%**	41%	41%	41%
Parser branch coverage	**57%**	29%	29%	29%
Rules line coverage	**63%**	42%	42%	42%
Rules branch coverage	**71%**	24%	24%	24%
Parser class coverage	**88%** (22/25)	80% (20/25)		
Rules class coverage	**100%** (20/20)	90% (18/20)		

Table 3: Application and package-level coverage statistics using the developer's functional tests, the full corpus with the full set of rules, and the full corpus with two reduced sets of rules. The highest value in a row is bolded. The final three columns are intentionally identical (see explanation in text).

Package	Line coverage >=	Line coverage >	Branch coverage >=	Branch coverage >
Classes in parser package	21/25	14/25	19/21	18/21
Classes in rules package	19/20	6/20	11/11	11/11

Table 4: When individual classes were examined, both line and branch coverage were always higher with the functional tests than with the corpus. This table shows the magnitude of the differences. >= indicates the number of classes that had equal or greater coverage with the functional tests than with the corpus, and > indicates just the classes that had greater coverage with the functional tests than with the corpus.

size of the original set. The first consisted of the first half of the rule set, which happened to consist primarily of verb-based patterns. The second consisted of the second half of the rule set, which corresponded roughly to the nominalization rules.

The last two columns of Table 3 show the package-level results. Overall, on a per-package basis, there were no differences in line or branch coverage when the data was run against the full rule set or either half of the rule set. (The identity of the last three columns is due to this lack of difference in results between the full rule set and the two reduced rule sets.) On a per-class level, we did note minor differences, but as Table 3 shows, they were within rounding error on the package level.

3.3 The third experiment: Coverage closure

In the third experiment, we looked at how coverage varies as increasingly larger amounts of the corpus are processed. This methodology is comparable to examining the closure properties of a corpus in a corpus linguistics study (see e.g. Chapter 6 of (McEnery and Wilson, 2001)) (and as such may be sensitive to the extent to which the contents of the corpus do or do not fit the sublanguage model). We counted cumulative line coverage as increasingly large amounts of the corpus were processed, ranging from 0 to 100% of its contents. The results for line coverage are shown in Figure 1. (The results for branch coverage are quite similar, and the graph is not shown.) Line coverage for the entire application is indicated by the thick solid line. Line coverage for the parser package is indicated by the thin solid line. Line coverage for the rules package is indicated by the light gray solid line. The broken line indicates the number of pattern matches—quantities should be read off of the right *y* axis.

The figure shows quite graphically the lack of effect on coverage of increasing the size of the corpus. For the entire application, the line coverage is 27% when an empty document has been read in, and 39% when a single sentence has been processed; it increases by one to 40% when 51 sentences have been processed, and has grown as high as it ever will—41%—by the time 1,000 sentences have been processed. Coverage at 191,478 sentences—that is, 3,947,200 words—is no higher than at 1,000 sentences, and barely higher, percentagewise, than at a single sentence.

An especially notable pattern is that the huge rise

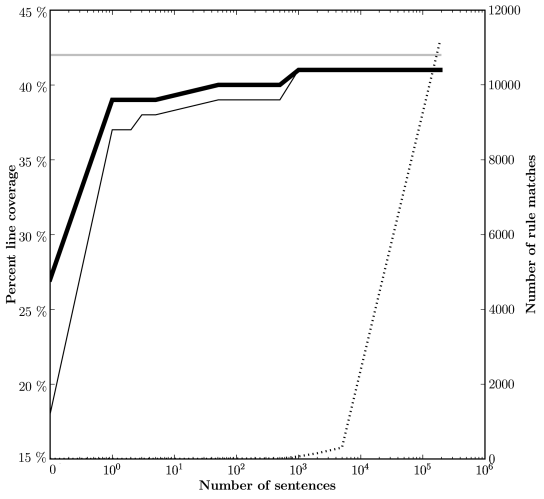

Figure 1: Increase in percentage of line coverage as increasing amounts of the corpus are processed. Left *y* axis is the percent coverage. The *x* axis is the number of sentences. Right *y* axis (scale 0-12,000) is the number of rule matches. The heavy solid line is coverage for the entire package, the thin solid line is coverage for the parser package, the light gray line is coverage for the rules package, and the broken line is the number of pattern matches.

in the number of matches to the rules (graphed by the broken line) between 5,000 sentences and 191K sentences has absolutely no effect on code coverage.

4 Discussion

The null hypothesis—that a synthetic test suite and a naturalistic corpus provide the same code coverage—is not supported by the data shown here. Furthermore, the widely, if implicitly, held assumption that a corpus would provide the best testing data can be rejected, as well. The results reported here are consistent with the hypothesis that code coverage for this application is not affected by the size of the corpus or by the size of the rule set, and that running it on a large corpus does not guarantee thorough testing. Rather, coverage is optimized by traditional software testing.

4.1 Related work

Although software testing is a first-class research object in computer science, it has received little attention in the natural language processing arena. A notable exception to this comes from the grammar engineering community. This has produced a body of publications that includes Oepen's work on test suite design (Oepen et al., 1998), Volk's work on test suite encoding (Volk, 1998), Oepen et al.'s work on the Redwoods project (Oepen et al., 2002), Butt and King's discussion of the importance of testing (Butt and King, 2003), Flickinger et al.'s work on "semantics debugging" with Redwoods data (Flickinger et al., 2005), and Bender et al.'s recent work on test suite generation (Bender et al., 2007). Outside of the realm of grammar engineering, work on testing for NLP is quite limited. (Cohen et al., 2004) describes a methodology for generating test suites for molecular biology named entity recognition systems, and (Johnson et al., 2007) describes the development of a fault model for linguistically-based ontology mapping, alignment, and linking systems. However, when most researchers in the NLP community refer in print to "testing," they do not mean it in the sense in which that term is used in software engineering. Some projects have publicized aspects of their testing work, but have not published on their approaches: the NLTK project posts module-level line coverage statistics, having achieved median coverage of 55% on 116 Python modules[2] and 38% coverage for the project as a whole; the MALLET project indicates on its web site that it encourages the production of unit tests during development, but unfortunately does not go into details of their recommendations for unit-testing machine learning code[3].

4.2 Conclusions

We note a number of shortcomings of code coverage. For example, poor coding conventions can actually inflate your line coverage. Consider a hypothetical application consisting only of the following, written as a single line of code with no line breaks: `if (myVariable == 1) doSomething elsif (myVariable == 2) doSomethingElse elsif (myVariable = 3) doYetAnotherThing` and a poor test suite consisting only of inputs that will cause `myVariable` to ever have the value `1`. The test suite will achieve 100% line coverage for

[2] `nltk.org/doc/guides/coverage`

[3] `mallet.cs.umass.edu/index.php/Guidelines_for_writing_unit_tests`

this application—and without even finding the error that sets `myVariable` to 3 if it is not valued 1 or 2. If the code were written with reasonable line breaks, code coverage would be only 20%. And, as has been noted by others, code coverage can not detect "sins of omission"—bugs that consist of the failure to write needed code (e.g. for error-handling or for input validation). We do not claim that code coverage is wholly sufficient for evaluating a test suite; nonetheless, it is one of a number of metrics that are helpful in judging the adequacy of a testing effort. Another very valuable one is the *found/fixed* or *open/closed graph* (Black, 1999; Baumgartner Jr. et al., 2007).

While remaining aware of the potential shortcomings of code coverage, we also note that the data reported here supports its utility. The developer-written functional tests were produced without monitoring code coverage; even though those tests routinely produced higher coverage than a large corpus of naturalistic text, *they achieved less than 60% coverage overall,* as predicted by Wiegers's work cited in the introduction. We now have the opportunity to raise that coverage via structured testing performed by someone other than the developer. In fact, our first attempts to test the previously unexercised code immediately uncovered two showstopper bugs; the coverage analysis also led us to the discovery that the application's error-handling code was essentially untested.

Although we have explored a number of dimensions of the space of the coverage phenomenon, additional work could be done. We used a relatively naive approach to rule ablation in the second experiment; a more sophisticated approach would be to ablate specific types of rules—for example, ones that do or don't contain slots, ones that do or don't contain regular expression operators, etc.—and monitor the coverage changes. (We did run all three experiments on a separate, smaller corpus as a pilot study; we report the results for the BioCreative II data set in this paper since that is the data for which the rules were optimized. Results in the pilot study were entirely comparable.)

In conclusion: natural language processing applications are particularly susceptible to emergent phenomena, such as interactions between the contents of a rule set and the contents of a corpus. These are especially difficult to control when the evaluation corpus is naturalistic and the rule set is data-driven. Structured testing does not eliminate this emergent nature of the problem space, but it *does* allow for controlled evaluation of the performance of your system. Corpora also are valuable evaluation resources: the *combination* of a structured test suite and a naturalistic corpus provides a powerful set of tools for finding bugs in NLP applications.

Acknowledgments

The authors thank James Firby, who wrote the functional tests, and Helen L. Johnson, who wrote the rules that were used for the BioCreative data. Steve Bethard and Aaron Cohen recommended Python coverage tools. We also thank the three anonymous reviewers.

References

William A. Baumgartner Jr., K. Bretonnel Cohen, Lynne Fox, George K. Acquaah-Mensah, and Lawrence Hunter. 2007. Manual curation is not sufficient for annotation of genomic databases. *Bioinformatics*, 23:i41–i48.

William A. Baumgartner Jr., Zhiyong Lu, Helen L. Johnson, J. Gregory Caporaso, Jesse Paquette, Anna Lindemann, Elizabeth K. White, Olga Medvedeva, K. Bretonnel Cohen, and Lawrence Hunter. In press. Concept recognition for extracting protein interaction relations from biomedical text. *Genome Biology*.

Emily M. Bender, Laurie Poulson, Scott Drellishak, and Chris Evans. 2007. Validation and regression testing for a cross-linguistic grammar resource. In *ACL 2007 Workshop on Deep Linguistic Processing*, pages 136–143, Prague, Czech Republic, June. Association for Computational Linguistics.

Rex Black. 1999. *Managing the Testing Process*.

Miriam Butt and Tracy Holloway King. 2003. Grammar writing, testing and evaluation. In Ali Farghaly, editor, *A handbook for language engineers*, pages 129–179. CSLI.

Geoffrey Chang, Christopher R. Roth, Christopher L. Reyes, Owen Pornillos, Yen-Ju Chen, and Andy P. Chen. 2006. Letters: Retraction. *Science*, 314:1875.

K. Bretonnel Cohen, Lorraine Tanabe, Shuhei Kinoshita, and Lawrence Hunter. 2004. A resource for constructing customized test suites for molecular biology entity identification systems. In *HLT-NAACL 2004 Workshop: BioLINK 2004, Linking Biological Literature, Ontologies and Databases*, pages 1–8. Association for Computational Linguistics.

Dan Flickinger, Alexander Koller, and Stefan Thater. 2005. A new well-formedness criterion for semantics debugging. In *Proceedings of the HPSG05 Conference*.

Lawrence Hunter, Zhiyong Lu, James Firby, William A. Baumgartner Jr., Helen L. Johnson, Philip V. Ogren, and K. Bretonnel Cohen. 2008. OpenDMAP: An open-source, ontology-driven concept analysis engine, with applications to capturing knowledge regarding protein transport, protein interactions and cell-specific gene expression. *BMC Bioinformatics*, 9(78).

Helen L. Johnson, K. Bretonnel Cohen, and Lawrence Hunter. 2007. A fault model for ontology mapping, alignment, and linking systems. In *Pacific Symposium on Biocomputing*, pages 233–244. World Scientific Publishing Company.

Cem Kaner, Hung Quoc Nguyen, and Jack Falk. 1999. *Testing computer software, 2nd edition*. John Wiley and Sons.

Maricel Kann, Yanay Ofran, Marco Punta, and Predrag Radivojac. 2006. Protein interactions and disease. In *Pacific Symposium on Biocomputing*, pages 351–353. World Scientific Publishing Company.

Martin Krallinger, Florian Leitner, and Alfonso Valencia. 2007. Assessment of the second BioCreative PPI task: automatic extraction of protein-protein interactions. In *Proceedings of the Second BioCreative Challenge Evaluation Workshop*.

Brian Marick. 1997. *The craft of software testing: subsystem testing including object-based and object-oriented testing*. Prentice Hall.

Steve McConnell. 2004. *Code complete*. Microsoft Press, 2nd edition.

Tony McEnery and Andrew Wilson. 2001. *Corpus Linguistics*. Edinburgh University Press, 2nd edition.

Greg Miller. 2006. A scientist's nightmare: software problem leads to five retractions. *Science*, 314:1856–1857.

Glenford Myers. 1979. *The art of software testing*. John Wiley and Sons.

S. Oepen, K. Netter, and J. Klein. 1998. TSNLP - test suites for natural language processing. In John Nerbonne, editor, *Linguistic Databases*, chapter 2, pages 13–36. CSLI Publications.

Stephan Oepen, Kristina Toutanova, Stuart Shieber, Christopher Manning, Dan Flickinger, and Thorsten Brants. 2002. The LinGO Redwoods treebank: motivation and preliminary applications. In *Proceedings of the 19th international conference on computational linguistics*, volume 2.

Martin Volk. 1998. Markup of a test suite with SGML. In John Nerbonne, editor, *Linguistic databases*, pages 59–76. CSLI Publications.