Natural language is a programming language: Applying natural language processing to software development

Michael D. Ernst[1]

1 **University of Washington Computer Science & Engineering, Seattle, WA, USA**
`mernst@cs.washington.edu`

Abstract

A powerful, but limited, way to view software is as source code alone. Treating a program as a sequence of instructions enables it to be formalized and makes it amenable to mathematical techniques such as abstract interpretation and model checking.

A program consists of much more than a sequence of instructions. Developers make use of test cases, documentation, variable names, program structure, the version control repository, and more. I argue that it is time to take the blinders off of software analysis tools: tools should use all these artifacts to deduce more powerful and useful information about the program.

Researchers are beginning to make progress towards this vision. This paper gives, as examples, four results that find bugs and generate code by applying *natural language processing* techniques to software artifacts. The four techniques use as input error messages, variable names, procedure documentation, and user questions. They use four different NLP techniques: document similarity, word semantics, parse trees, and neural networks.

The initial results suggest that this is a promising avenue for future work.

1998 ACM Subject Classification D.2 Software Engineering, F.3.2 Semantics of Programming Languages, I.2.7 Natural Language Processing

Keywords and phrases natural language processing, program analysis, software development

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.4

1 Introduction

What is software? A reasonable definition — and the one most often adopted by the programming language community — is: a sequence of instructions that perform some task. This definition accommodates the programmer's view of source code and the machine instructions that the CPU executes. Furthermore, this definition enables formalisms: the execution model of the machine, and the meaning of every instruction, can be mathematically defined, for example via denotational semantics or operational semantics. By combining the meanings of each instruction, the meaning of a program can be induced.

This perspective leads to powerful static analyses, such as symbolic analysis, abstract interpretation, dataflow analysis, type checking, and model checking. Equally important and challenging theoretically — and probably more important in practice — are dynamic analyses that run the program and observe its behavior. These are at the heart of techniques such as testing, error detection and localization, debugging, profiling, tracing, and optimization.

Despite the successes of viewing a program as a sequence of instructions — essentially, of treating a program as no more than an AST (abstract syntax tree) — this view is limited and foreign to working programmers, who should be the focus of research in programming

SNAPL: The 2nd Summit on Advances In Programming Languages.
Editors: Ras Bodik and Shriram Krishnamurthi; Article No. 4; pp. 4:1–4:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

languages. Developers make use of test cases, documentation, variable names, program structure, the version control repository, the issue tracker, conversations, user studies, analyses of the problem domain, executions of the program, and much more. The very successes of formal analysis may have blinded the research community to the bigger picture. In order to help programmers, and even to provide the program specifications that are essential to formal analysis, software analysis tools need to analyze all the artifacts that developers create. Tools that analyze the whole program will deduce more powerful and useful information about the program than tools that view just one small slice of it. These non-AST aspects of the program are also good targets for generation or synthesis approaches, especially since developers usually encode information redundantly: the information can be recovered from other (formal or informal) sources of information.

This paper focuses on one part of this vision: analysis of the natural language that is embedded in the program. In order to provide inspiration for further research, the paper discusses four initial results that find bugs and generate code by applying natural language processing techniques to software artifacts. The four techniques use as input error messages, variable names, procedure documentation, and user questions. They use four different NLP techniques: document similarity, word semantics, parse trees, and neural networks. In many cases, they produce a formal artifact from an informal natural language input. The initial results show the promise of applying NLP to programs.

This paper is organized as follows. First, section 2 puts the use of NLP in the context of previous work that uses non-standard sources of specifications for formal analysis. In other words, section 2 shows how using NLP to produce specifications can be viewed as the continuation of an existing line of research. The following four sections present four different approaches to applying natural language processing to English text that is associated with a program. Each one addresses a different problem, uses a different source of natural language, and applies a different natural language technique to the English to solve the problem. The following table overviews the four approaches.

		Problem	NL source	NLP technique
§3	Analyze existing	inadequate diagnostics	error messages	document similarity
§4	code to find bugs	incorrect operations	variable names	word semantics
§5	Generate	missing tests	code comments	parse trees
§6	new code	unimplemented functionality	user questions	translation

These few examples cover only a small number of problems, sources of natural language, and NLP techniques. Other researchers can take inspiration from these examples in order to pursue further research in this area. Section 7 discusses how researchers are already doing related work, via text analysis, machine learning, and other approaches.

2 Background: Mining specifications

Students sometimes ask whether a program is correct[1], but such a question is ill-posed. A program is never correct or incorrect; rather, the program either satisfies a specification or fails to satisfy a specification. It is no more sensible to ask whether a program is correct, without stating a specification, than to ask whether the answer is 42, without stating the question to be answered.

[1] There are many other important questions to be asked about a program beyond correctness. Does it fulfill a need in the real world? Is it usable? Is it reliable? Is it maintainable?

Many tasks, such as verification and bug detection, require a specification that expresses what the program is supposed to do. As a result, many papers start out by assuming the existence of a program and a specification; given these artifacts, the paper presents a program analysis technique. Unfortunately, most programs do not come with a formal specification. Furthermore, programmers are reluctant to write them, because they view the cost of doing so as greater than the benefit. Researchers and tool makers need to make specifications easier to write, and they need to create tools that provide value to workaday programmers. Until that happens, there is still an urgent need for specifications, in order to apply the research and tools that have been created.

One effective approach is to mine specifications — that is, to infer them from artifacts that programmers *do* create. Programmers embed rich information in the artifacts that they create. *Program analysis tools should take advantage of all the information in programs, not just the AST.* Too often, this is not done. For example, before formally verifying a program, the program is always tested, because testing is a more cost-effective way to find most errors. However, the formal verification process generally ignores all the effort that was put into testing, the test suites that were created, and the knowledge that was gained. This is a missed opportunity, in part caused by a parochial blindness toward "non-formal" artifacts.

Another way to express this intuition is to contrast two different views of a software artifact. Traditionally, programming language researchers have viewed it as an engineered artifact with well-understood semantics that is amenable to formal analysis. An alternative view is as a natural object with unknown properties that has to be probed and measured in order to understand it. These two perspectives contrast an engineer's blueprint with a natural scientist's explorations of the world. Considering a program as a natural object enables many powerful analyses, such as machine learning over executions, version control history analysis, prediction of upgrade safety, bug prediction, warning prioritization, and program repair.

As one example, consider specification mining: machine learning of likely specifications from executions. This technique transforms the implicit specifications that the programmer has embedded into a test suite, into a formal specification. A tool that performs this task is the Daikon invariant detector [16, 17, 18]; other tools also exist [2, 24, 41, 57, 9, 7, 8]. The software developer runs the program, and Daikon observes the values that the program computes. Daikon generalizes over the values via machine learning, in particular using a generate-and-check approach augmented by static and dynamic analyses and optimizations, because prior learning approaches had limitations that prevented them from being applied to this domain. The output is properties such as

- `x > abs(y)`
- `x = 16*y + 4*z + 3`
- array `a` contains no duplicates
- for each node `n`, `n = n.child.parent`
- graph `g` is acyclic

Like any good machine learning algorithm, the technique is unsound, incomplete, and *useful*. It is unsound because these are likely invariants: they were true over all executions and passed statistical likelihood tests, but there is no guarantee that they will be true during all possible future executions. It is incomplete because every machine learning algorithm has a bias or a grammar that limits its inferences. Nonetheless, it is useful. Some uses do not require soundness, such as optimization or bug-finding. Humans are known to make good use of imperfect information. The likely invariants can be used as goals for a verifier, yielding a sound system. Automatically-generated partial information is better than none at all. In practice, the inference process is surprisingly effective: the invariants are overwhelmingly correct, even when generalizing from little execution data.

Just as it is useful to process test suites to create formal artifacts, it is also useful to process natural language to create formal artifacts. The following sections give some examples.

3 Detection of inadequate diagnostic messages

Software configuration errors (also known as misconfigurations) are errors in which the software code and the input are correct, but the software does not behave as desired because an incorrect value is used for a configuration option [61, 58, 54, 56]. Diagnostic messages are often the sole data source available to a developer or user. Unfortunately, many configurable software systems have cryptic, hard to understand, or even misleading diagnostic messages [58, 28], which may waste up to 25% of a software maintainer's time [6]. We have built a tool, ConfDiagDetector [62], that tells a developer, before their application is fielded, whether the diagnostic messages are adequate.

More concretely, if a user supplies a wrong configuration option such as `-port_num=100.0`, the software may issue a hard-to-diagnose error message such as "unexpected system failure" or "unable to establish connection". Our goal is to detect such problems before shipping the code, so that the developer can substitute a better message, such as "`-port_num` should be an integer".

ConfDiagDetector combines two main ideas: configuration mutation and NLP text analysis. ConfDiagDetector works by injecting configuration errors into a configurable system, observing the resulting failures, and using NLP text analysis to check whether the software issues an informative diagnostic message relevant to the root-cause configuration option (the one related to the injected configuration error). If not, ConfDiagDetector reports the diagnostic message as inadequate.

ConfDiagDetector considers a diagnostic message as adequate if contains the mutated option name or value [29, 58], or if its meaning is semantically similar to the manual description of that configuration option. For example, if the `-fnum` option was mutated and its manual description says "Sets number of folds for cross-validation", then the diagnostic message "Number of folds must be greater than 1" is adequate.

Classical document similarity work uses TF-IDF (term frequency – inverse document frequency) to convert each document into a real-valued vector, then uses vector cosine similarity. This approach does not work well on very short documents, such as diagnostic messages, so ConfDiagDetector instead uses a different technique that counts similar words [36].

In a case study, ConfDiagDetector reported 25 missing and 18 inadequate messages in four open-source projects: Weka, JMeter, Jetty, and Derby. A validation by three programmers indicated that ConfDiagDetector has a 0% false negative rate and a 2% false positive rate on this dataset. This is a significant improvement over the previous best tool, which had a 16% false positive rate.

This approach differs from configuration error diagnosis techniques such as dynamic tainting [4], static tainting [43, 44], and Chronus [54] that troubleshoot an exhibited error, rather than proactively detecting inadequate diagnostic messages. It also differs from software diagnosability improvement techniques such as PeerPressure [53], RangeFixer [55], ConfErr [29], Spex-INJ [58], and EnCore [60] that require source code, a usage history, or OS-level support.

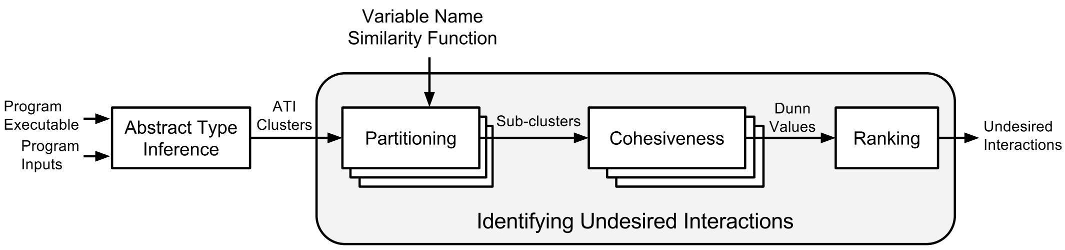

Figure 1 Ayudante architecture.

4 Identifying undesired variable interactions

A common programming mistake is for incompatible variables to interact, e.g., storing euros in a variable that should hold dollars, or using an array index with the wrong array. When a programmer commits an error, such as writing `totalPrice = itemPrice + shippingDistance;`, the compiler issues no warning because the two variables have the same programming language type, such as `int`. However, a human can tell that the abstract types are different, based on the variable names that the programmer chose.

We have developed an approach to detect such undesired interactions [52]. The approach clusters related variables, twice, using two different mechanisms. Natural language processing identifies variables with related names that may have related semantics. Abstract type inference identifies variables that interact with each other, which the programmer has treated as related. (For example, if the programmer wrote `x < y`, then the programmer must view `x` and `y` as having the same abstract type.) Any discrepancies between these two clusterings — that is, any inconsistency between variable names and program operations — may indicate a programming error, such as a poorly-named variable or an incorrect program operation.

Ayudante clusters variable names by tokenizing each variable name into dictionary words, computing word similarity based on WordNet or edit distance, and then arithmetically combining word similarity into variable name similarity. These variable name similarities can be treated as distances by a clustering algorithm. When a single ATI cluster can be split into two distinct variable-name clusters, it is treated as suspicious and presented to a user. Figure 1 shows the high-level architecture.

Abstract type inference can be computed statically [5, 37] or dynamically [22]; our tool, Ayudante, uses the dynamic approach, which is more precise in practice.

In an experiment, Ayudante's top-ranked report about the grep program indicated a interaction in grep that was likely undesired, because it discards information.

Previous work showed that reusing reusing identifier names is error-prone [32, 14, 3] and proposed identifier naming conventions [46, 30]. Languages like Ada and F# support a notation for units of measure. Our tokenization of variable names outperforms previous work [31, 21].

5 Generation of test oracles

Programmers are resistant to writing formal specifications or test oracles. Manually-written test suites often neglect important behavior. Automatically-generated test suites, on the other hand, lack test oracles that verify whether the observed behavior is correct. We have implemented a technique that automatically creates test oracles from something that programmers already write: code comments. In particular, it is standard practice for Java

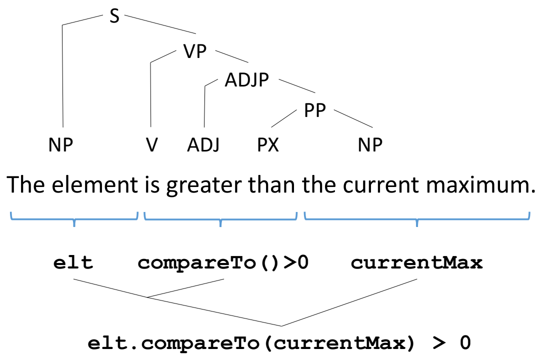

Figure 2 Parsing a sentence and unparsing into an assertion.

programmers to write Javadoc comments; IDEs even automatically insert templates for them.

We have built a tool, Toradocu [19], that converts English comments into assertions. For example, given

```
/** @throws IllegalArgumentException if the
 * element is not in the list and is not
 * convertible. */
void myMethod(Object element) { ... }
```

Toradocu might determine that `myMethod` should throw the exception iff
`( !allFoundSoFar.contains(element) && !canConvert(element) ).`

The intuition behind the technique is that when a sentence describes program behaviors, its nouns correspond to objects or values, and its verbs correspond to operations. This enables translation between English and code.

Toradocu works in the following steps.

1. Toradocu determines the nouns and verbs in a sentence from a Javadoc `@param`, `@return`, or `@throws` clause. It does so using the Stanford Parser, which yields a parse tree, grammatical relations, and cross-references. Toradocu uses pre- and post-processing to handle challenges such as the fact that the natural language is often not a well-formed sentence, it may use code snippets as nouns/verbs, and referents may be implicit.
2. Toradocu matches each noun/subject in the sentence to a code element from the program. It uses both pattern matching and lexical similarity to identifiers, types, and documentation.
3. Toradocu matches each verb/predicate to a Java element.
4. Toradocu reverses the parsing step: it recombines the identified Java elements, according to the parse tree of the original English sentence. The result is an assert statement.

Figure 2 gives an example.

In an experiment on 941 programmer-written Javadoc specifications, Toradocu achieved 88% precision and 59% recall in translating them to executable assertions. Toradocu can be tuned to favor either precision or recall.

Toraducu can automatically instrument test suites. Currently, automatic test generation tools have to guess whether a generated test fails or passes. Toradocu improved the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduced EvoSuite's false positive test failures by 33%.

Previously, test generation tools used heuristics to guess whether an exception was expected or unexpected [12, 13, 38, 39]. Property-based techniques that are similar to or can benefit from our approach include cross-checking oracles [10], metamorphic testing [11],

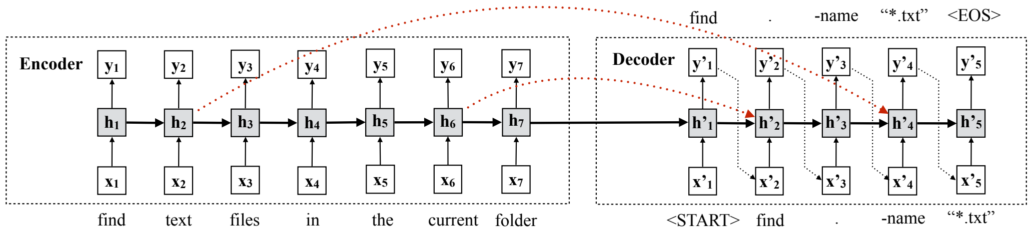

Figure 3 A sequence-to-sequence neural network translation model, applied to English and bash commands. The encoder reads the natural language description and passes its final hidden state to the decoder. The decoder takes the encoder's final hidden state and generates the output starting form a special symbol `<START>`. Notice that each decoder input symbol is the output symbol from the previous step. As is traditional, boxes are labeled by their outputs; for example, the lowest, leftmost box takes as input x_t (= "find") and applies I, producing as output $\mathbf{x}_t$. The red dotted lines mark the word alignments learned via the attention mechanism. While the neural network computes an alignment score for each pair of encoder hidden state and decoder hidden state, we illustrate only the alignments with high scores for readability.

and symmetric testing [20]. Previous work has used pattern-matching to extract simple properties, like whether a variable is intended to be non-null or nullable, from natural language documentation [51, 50, 49]; our approach is more general because it uses more sophisticated natural language processing techniques.

6 Generating code from natural-language specifications

The job of a software developer includes determining the customer's requirements and implementing a program that satisfies them. Part of this job is translating from a (usually informal) specification into source code.

One of the great successes of natural language processing is translation: for example, converting the English sentence "My hovercraft is full of eels" into the Spanish sentence "Mi aerodeslizador está lleno de anguilas." Recently, recurrent neural networks (RNNs) have come to dominate machine translation. The neural network is trained on a great deal of known correct data (English–Spanish pairs), and the network's input, hidden, and output functions are inferred using probability maximization.

If this approach works well for natural language, why shouldn't it work for programming languages? In other words, why can't we create a program — or, at least, get an initial draft — from natural language?

We have applied this approach to convert English specifications of file system operations into bash commands [33]. Figure 3 shows a concrete example. We trained the RNN on 5,000 ⟨text, bash⟩ pairs that were manually collected from webpages such as Stack Overflow and bash tutorials. This domain includes 17 file system utilities, more than 200 flags, 9 types of open-vocabulary constants, and nested command structures such as pipelines, command substitution, and process substitution. Our system Tellina's top-1 and top-3 accuracy, for the structure of the command, was 69% and 80%.

No natural language technique will achieve perfect accuracy, due to the underlying machine learning algorithms. Tellina produces correct results most of the time, but produces incorrect results the rest of the time.[2] It is an important and interesting empirical question whether such a system that is useful in practice to programmers. In a controlled human experiment, programmers using Tellina spent statistically significantly less time ($p < .01$) while completing more file system tasks ($p < .1$). Even when Tellina's output was not perfect, it often informed the programmer about a command-line flag that the programmer didn't know about.

The most closely related work is in neural machine translation, which proposed both sequence-to-sequence learning with neural nets [48] and the attention mechanism [35]. Previous work on semantic parsing has translated natural language to a formal representation [59, 40], though one simpler than bash. Previous work on translating natural language to DSLs has also focused on simpler languages: if-this-then-that recipes [42], regular expressions [34], and text editing and flight queries [15].

7 Discussion

A minority of a software development team's is spent writing and changing the program, as opposed to participating in other activities, such as gathering requirements, design, documentation, and communicating with peers and stakeholders. Even when interacting with the program, a minority of a programmer's time is spent editing the programming language constructs in the source code, as opposed to testing, documenting, debugging, and reading it to understand it.

Researchers in software engineering and programming languages can find the most important challenges, do the most relevant work, and have the most impact by recognizing the needs of software developers. The programming language itself is an important but small part of this.

This paper advocates using natural language processing to analyze the textual parts of a program, in addition to the machine operations or AST that form its mathematical or operational core. Even the program including its natural language (the focus of this paper) still represents a minority of the concerns of a software developer! This paper focused on it because it is an important domain that permits use of a coherent set of research techniques. These techniques can apply ideas from both natural language processing and program analysis, and crucially, they can produce formal, executable specifications that feed back into many techniques that require specifications to express program semantics.

Our point of view is related to many previous lines of work. Previous researchers have applied pattern-matching or machine learning techniques (in some cases including NLP techniques), to software development artifacts that include the (formal) program, natural language in it, its tests, and its development history. We acknowledge their achievements, which have enabled and/or inspired our own.

The idea of analyzing the text that accompanies a program is not new. Up to now, much of this textual processing has been pattern-matching [49] rather than NLP. The same is true of many other approaches to processing program text, as described earlier. We believe that use of NLP will enable these techniques to become more general and achieve better results.

Statistical models can be used to model program text in similar ways to modeling natural language. Hindle et al. [26] hypothesize that "what people write and say is largely regular and predictable". This regularity is captured by n-gram models that capture how often a

[2] Classifying the usefulness of Tellina's output is not clear-cut. Even Tellina's correct results may not be perfect, and even its incorrect results can be helpful to programmers.

given sequence of n tokens occurs. This work ignores comments and applies these models to the executable program statements and expressions. The authors proposed that N-gram models can be used for code completion (such as stylized `for` loops). Subsequent work applied n-gram models to predicting common variable names and whitespace conventions [1]. Neither approach captures semantics other than incidentally by correlation, and neither was evaluated in terms of whether it would help programmers.

Another line of work focuses on creating the building blocks that from which NLP semantics could be obtained by future tools. Pollock and colleagues show how common variable-name patterns can be analyzed to assign a part of speech to each word that makes up the variable name [23], how rules and heuristics can match verbs to semantically-similar words by examining both code and comments [27], and how to mine abbreviation expansions such as "num" vs. "number" in variable names [25]. They also show how to generate summary comments for code [47], which is the dual of our goal of transforming less-formal into more-formal artifacts.

The JSNice system [45] represents a program AST in relational form for input to a learner. Given libraries/APIs that have known types and commonly-associated names, names and types can be inferred for new clients of those programs. This can regularize existing programs or suggest names for identifiers in new programs. It can also suggest types, without doing a standard type analysis. This work is notable for its uptake by industry. The variable names do not affect program semantics, the types are optional, and the compiler warnings can be suppressed; nonetheless, JSNice is useful in improving code style and gradually adding types to JavaScript code.

As the above examples show, natural language processing (NLP) is just one form of machine learning. NLP is applicable to the textual aspects of a program, such as messages, variable names, code comments, and discussions. Other types of data mining and machine learning can be applied to natural language in the text or to other artifacts, such as executions (e.g., section 2), bug reports, version control history, developer conversations, and much more. The ideas presented in this paper could be extended to those other domains as well.

8 Analyzing the entire program

A program is more than source code, because a programming language — and more importantly, the programming system that surrounds it — is more than just a mathematical abstraction. In order to manage and understand the complexity of their programs, software developers embed important, useful information in test suites, error messages, manuals, variable names, code comments, and specifications. By paying attention to these rich sources of information, we can produce better software analysis tools and make programmers more productive. In addition to laying out this vision, this paper has overviewed a few concrete steps toward the vision: projects in which this extra information has proved useful. Many more opportunities exist, and I urge the community to grasp them.

Acknowledgements. This is joint work with Arianna Blasi, Juan Caballero, Sergio Delgado Castellanos, Alberto Goffi, Alessandra Gorla, Xi Victoria Lin, Deric Pang, Mauro Pezzè, Irfan Ul Haq, Kevin Vu, Chenglong Wang, Luke Zettlemoyer, and Sai Zhang. The reviewers provided helpful comments that improved the paper.

References

1 Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 281–293, Hong Kong, November 2014.

2 Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL 2002: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 2002.

3 Venera Arnaoudova, Laleh Eshkevari, Rocco Oliveto, Yann-Gael Gueheneuc, and Giuliano Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *ICSM 2010: 26th IEEE International Conference on Software Maintenance*, pages 1–5, Timişoara, Romania, September 2010.

4 Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, pages 281–286, Boston, Massachusetts, 2008.

5 Henry Baker. Unify and conquer (garbage, updating, aliasing, ...). In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 218–226, Nice, France, June 1990.

6 Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila A. Takayama, and Madhu Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *CSCW 2004: Computer Supported Cooperative Work*, pages 388–395, Chicago, IL, USA, November 2004.

7 Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ICSE 2013, Proceedings of the 35th International Conference on Software Engineering*, pages 252–261, San Francisco, CA, USA, May 2013.

8 Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pages 468–479, Hyderabad, India, June 2014.

9 Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–277, Szeged, Hungary, September 2011.

10 Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pages 931–942, Hyderabad, India, June 2014.

11 Tsong Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *STEP'03, Proceedings of the 11th International Workshop on Software Technology and Engineering Practice*, pages 94–100, 2003.

12 Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

13 Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE 2005, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 2005.

14 Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

15 Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, 2016.

16 Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 1999.

17 Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

18 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

19 Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, Saarbrücken, Genmany, July 2016.

20 Arnaud Gotlieb. Exploiting symmetries to test programs. In *ISSRE'03, Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 365–375, 2003.

21 Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *WCRE*, 2012.

22 Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 2006.

23 Samir Gupta, Sana Malik, Lori Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *ICPC 2013: Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 3–12, San Francisco, CA, USA, May 2013.

24 Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE 2002, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 2002.

25 Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR 2008: 5th Working Conference on Mining Software Repositories*, pages 79–88, Leipzig, Germany, May 2008.

26 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE 2011, Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, Zürich, Switzerland, June 2012.

27 Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *MSR 2013: 10th Working Conference on Mining Software Repositories*, pages 377–386, San Francisco, CA, USA, May 2013.

28 Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in Linux and eCos. In *VaMoS 2012: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 149–155, Leipzig, Germany, January 2012.

29 Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN 2008: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 157–166, Achorage, AK, USA, June 2008.

30 Doug Klunder. Naming conventions (Hungarian). Internal Microsoft document, January 18, 1988.

31 Dawn Lawrie, Christopher Morrell, and Dave Binkley. Normalizing source code vocabulary. In *WCRE 2010: 2010 17th Working Conference on Reverse Engineering*, pages 3–12, Beverly, MA, USA, October 2010.

32 Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, December 2007.

33 Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2017.

34 Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 1918–1923, 2016.

35 Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015.

36 Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI 2006: Proceedings of the 21st National Conference on Artificial Intelligence*, pages 775–780, Boston, MA, USA, July 2006.

37 Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

38 Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 2005.

39 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 2007.

40 Panupong Pasupat and Percy Liang. Inferring logical forms from denotations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

41 Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG 2003: Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, September 2003.

42 Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 878–888, Beijing, China, July 2015.

43 Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*, pages 193–202, Lawrence, KS, USA, November 2011.

44 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140, Waikiki, Hawaii, USA, May 2011.

45 Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "Big Code". In *POPL 2015: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Sym-*

posium on Principles of Programming Languages, pages 111–124, Mumbai, India, January 2015.

46 Charles Simonyi. Hungarian notation. `https://msdn.microsoft.com/en-us/library/aa260976%28VS.60%29.aspx`.

47 Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE 2010: Proceedings of the 25th Annual International Conference on Automated Software Engineering*, pages 43–52, Antwerp, Belgium, September 2010.

48 Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

49 Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: Bugs or bad comments?*/. In *SOSP 2007, Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 145–158, Stevenson, WA, USA, October 2007.

50 Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 11–20, Waikiki, Hawaii, USA, May 2011.

51 Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *ICST 2012: Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 260–269, Montreal, Canada, April 2012.

52 Irfan Ul Haq, Juan Caballero, and Michael D. Ernst. Ayudante: Identifying undesired variable interactions. In *WODA 2015: 13th International Workshop on Dynamic Analysis*, pages 8–13, Pittsburgh, PA, USA, October 2015.

53 Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI 2004: USENIX 6th Symposium on OS Design and Implementation*, pages 245–257, San Francisco, CA, USA, December 2004.

54 Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI 2004: USENIX 6th Symposium on OS Design and Implementation*, pages 77–90, San Francisco, CA, USA, December 2004.

55 Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE 2011, Proceedings of the 34th International Conference on Software Engineering*, pages 58–68, Zürich, Switzerland, June 2012.

56 Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. Range fixes: Interactive error resolution for software configuration. *IEEE Transactions on Software Engineering*, 41(6):603–619, June 2014.

57 Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE 2006, Proceedings of the 28th International Conference on Software Engineering*, pages 282–291, Shanghai, China, May 2006.

58 Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172, Cascais, Portugal, 2011.

59 Luke S. Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 678–687, 2007.

60 Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhangand Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS 2014: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–700, Houston, TX, USA, March 2014.

61 Sai Zhang and Michael D. Ernst. Which configuration option should I change? In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pages 152–163, Hyderabad, India, June 2014.

62 Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 12–23, Baltimore, MD, USA, July 2015.