Natural language is a programming language

Michael D. Ernst
UW CSE

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
Questions about software

• How many of you have used software?
• How many of you have written software?
What is software?
What is software?

- A sequence of instructions that perform some task
What is software?

An engineered object amenable to formal analysis
• A sequence of instructions that perform some task
**Formalizations**

\[ P \in \text{Program} ::= \text{Class, ClassId, Expr} \]

\[ \text{Cls} \in \text{Class} ::= \text{class ClassId<\text{TVarId}, Fs} \in \text{Fields} \]

\[ {^sT} \in {^sType} ::= {^sNType} | \text{TVarId} \]

\[ {^sN} \in {^sNType} ::= \text{OM ClassId<} {^sType} \]

\[ \text{u} \in \text{OM} ::= \]

\[ \text{mt} \in \text{Meth} ::= \]

\[ \text{MethSig} ::= \]

\[ w \in \text{Purity} ::= \]

\[ e \in \text{Expr} ::= \]

\[ {^sT} \in {^sEnv} ::= \]

\[ \begin{align*}
\text{GT-Read} & \quad \Gamma \vdash e_0 : N_0 \quad N_0 = u_0 \ C_0 < > \\
\end{align*} \]

\[ \begin{align*}
\text{GT-Update} & \quad \Gamma \vdash e_0.f : N_0 \triangleright f Type(C_0, f) \\
\end{align*} \]

\[ \begin{align*}
\text{DYN} & \quad \text{Free}({^sT}) \subseteq \text{dom}(C_N) \\
\end{align*} \]

\[ \begin{align*}
\text{Dyn}({^sT}, i, {^rT}, (X^r T'; -)) = {^sT}[i'/\text{this}, i'/'\text{peer}, i'/\text{rep}, \text{any}_a/\text{any}_u, {^rT}/X, {^rT'}/X'] \\
\end{align*} \]
What is software?

• A sequence of instructions that perform some task
What is software?

- A sequence of instructions that perform some task
- Test cases
- Version control history
- Issue tracker
- Documentation
- ...

How should it be analyzed?
Programming

- Requirements
- Discussions
- User stories
- Specifications
- Models
- Issue tracker
- Version control
- Models
- Process
- Programs
- Architecture
- Tests
- Documentation
Programming

Requirements

Issue tracker

Documentation

Test

Version control

Architectures

Process

Discussion

User stories

Specifications

Models

Issue tracker

PL

Structure

Output strings

Variable names

Documentation
Analysis of a natural object

• Machine learning over executions
• Version control history analysis
• Bug prediction
• Upgrade safety
• Prioritizing warnings
• Program repair
Specifications are needed; Tests are available but ignored

- **Specs are needed.** Many papers start: “Given a program and its specification...”
- **Tests are ignored.** Formal verification process:
  - Write the program
  - Test the program
  - Verify the program, *ignoring* testing artifacts

**Observation:** Programmers embed semantic info in tests

**Goal:** translate tests into specifications

**Approach:** machine learning over executions
Dynamic detection of likely invariants

• Observe values that the program computes
• Generalize over them via machine learning
• Result: invariants (as in `asserts` or specifications)
  • \( x > \text{abs}(y) \)
  • \( x = 16*y + 4*z + 3 \)
  • array \( a \) contains no duplicates
  • for each node \( n \), \( n = n.child.parent \)
  • graph \( g \) is acyclic
• Unsound, incomplete, and useful
Programming

- Requirements
- Discussions
- Models
- Specifications
- Programs
  - Issue tracker
  - Version control
  - Documentation
  - Process
  - Architecture
  - Tests
  - PL
  - Structure
  - Documentation
  - Output strings
  - Variable names
Programming

Requirements

Issue tracker

Documentation

Version control

Process

Tests

Discussions

Models

Specifications

Variables names

PL

Structure

Documentation

Output strings

Issue tracker

Architecture

Formalizations

Variable names
Programming

- Requirements
- Discussions
- Models
- Specifications
- Programs
- Issue tracker
- Documentation
- Version control
- Process
- Tests
- Architecture
- PL
- Structure
- Documentation
- Output strings
- Variable names
- Formalizations

Variable names
Applying NLP to software engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>NL sources</th>
<th>NLP techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>inadequate diagnostics</td>
<td>error messages</td>
<td>document similarity</td>
</tr>
<tr>
<td>incorrect operations</td>
<td>variable names</td>
<td>word semantics</td>
</tr>
<tr>
<td>missing tests</td>
<td>code comments</td>
<td>parse trees</td>
</tr>
<tr>
<td>unimplemented functionality</td>
<td>user questions</td>
<td>translation</td>
</tr>
</tbody>
</table>
## Applying NLP to software engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>NL sources</th>
<th>NLP techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>inadequate diagnostics</td>
<td>error messages</td>
<td>document similarity</td>
</tr>
<tr>
<td>incorrect operations</td>
<td>variable names</td>
<td>word semantics</td>
</tr>
<tr>
<td>missing tests</td>
<td>code comments</td>
<td>parse trees</td>
</tr>
<tr>
<td>unimplemented functionality</td>
<td>user questions</td>
<td>translation</td>
</tr>
</tbody>
</table>

[ISSTA 2015]
Inadequate diagnostic messages

**Scenario:** user supplies a wrong configuration option
--port_num=100.0

**Problem:** software issues an unhelpful error message
- “unexpected system failure”
- “unable to establish connection”

Hard for end users to diagnose

**Goal:** detect such problems *before* shipping the code
- Better message: “--port_num should be an integer”
Challenges for proactive detection of inadequate diagnostic messages

• How to *trigger a configuration error*?

• How to *determine the inadequacy* of a diagnostic message?
ConfDiagDetector’s solutions

• **How to trigger a configuration error?**
  - Configuration mutation + run system tests
    
    ![Configuration mutation + run system tests](image)
    
    configuration + system tests → failed tests ≈ triggered errors
    (We know the root cause.)

• **How to determine the inadequacy of a diagnostic message?**
  - Use a **NLP technique** to check its semantic meaning
    
    ![Similar semantic meanings?](image)
    
    Diagnostic messages output by failed tests → User manual
    (Assumption: a manual, webpage, or man page exists.)
When is a message adequate?

• Contains the mutated option name or value [Keller’08, Yin’11]

  Mutated option:
  
  --percentage-split

  Diagnostic message:
  
  “the value of percentage-split should be > 0”

• Similar semantic meaning as the manual description

  Mutated option:
  
  --fnum

  Diagnostic message:
  
  “Number of folds must be greater than 1”

  User manual description of --fnum:
  
  “Sets number of folds for cross-validation”
Classical document similarity: TF-IDF + cosine similarity

1. Convert document into a real-valued vector
2. Document similarity = vector cosine similarity

• Vector length = dictionary size, values = term frequency (TF)
  • Example: [2 classical, 8 document, 3 problem, 3 values, ...]
• Problem: frequent words swamp important words
• Solution: values = TF x IDF (inverse document frequency)
  • IDF = log(total documents / documents with the term)

Problem: does not work well on very short documents
Text similarity technique [Mihalcea’06]

The documents have similar semantic meanings if many words in them have similar meanings.

Example:

1. Remove all stop words.
2. For each word in the diagnostic message, try to find similar words in the manual.
3. Two sentences are similar, if “many” words are similar between them.
Results

• Reported 25 missing and 18 inadequate messages in Weka, JMeter, Jetty, Derby

• Validation by 3 programmers:
  • 0% false negative rate
    • Tool says message is adequate, humans say it is inadequate
  • 2% false positive rate
    • Tool says message is inadequate, humans say it is adequate
    • Previous best: 16%
Related work

Configuration error diagnosis techniques

- Dynamic tainting [Attariyan’08], static tainting [Rabkin’11], Chronus [Whitaker’04]

Troubleshooting an exhibited error rather than detecting inadequate diagnostic messages

Software diagnosability improvement techniques

- PeerPressure [Wang’04], RangeFixer [Xiong’12], ConfErr [Keller’08] and Spex-INJ [Yin’11], EnCore [Zhang’14]

Requires source code, usage history, or OS-level support
# Applying NLP to software engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>NL sources</th>
<th>NLP techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>inadequate diagnostics</td>
<td>error messages</td>
<td>document similarity</td>
</tr>
<tr>
<td>incorrect operations</td>
<td>variable names</td>
<td>word semantics</td>
</tr>
<tr>
<td>missing tests</td>
<td>code comments</td>
<td>parse trees</td>
</tr>
<tr>
<td>unimplemented functionality</td>
<td>user questions</td>
<td>translation</td>
</tr>
</tbody>
</table>

[WODA 2015]
Undesired variable interactions

```c
int totalPrice;
int itemPrice;
int shippingDistance;

totalPrice = itemPrice + shippingDistance;
```
Undesired variable interactions

```java
int totalPrice;
int itemPrice;
int shippingDistance;

totalPrice = itemPrice + shippingDistance;
```

- The compiler issues no warning
- A human can tell the abstract types are different

Idea:
- Cluster variables based on words in variable names
- Cluster variables based on usage in program operations

Differences indicate bugs or poor variable names
Undesired interactions

distance  itemPrice  tax_rate

miles  shippingFee  percent_complete
Undesired interactions

\[ \text{distance} \leftrightarrow \text{itemPrice} \quad \text{tax\_rate} \]
\[ \text{itemPrice} + \text{distance} \]

\[ \text{miles} \quad \text{shippingFee} \quad \text{percent\_complete} \]
Undesired interactions

Program types don’t help
Undesired interactions

Language indicates the problem
Variable clustering

Cluster based on interactions: operations
Variable clustering

Cluster based on language: variable names
Variable clustering

Cluster based on interactions: operations

Cluster based on language: variable names

Actual algorithm:
1. Cluster based on operations
2. Sub-cluster based on names
3. Rank an operation cluster as suspicious if it contains well-defined name sub-clusters
Clustering based on operations

Abstract type inference [ISSTA 2006]

```java
int totalCost(int miles, int price, int tax) {
    int year = 2016;
    if ((miles > 1000) && (year > 2000)) {
        int shippingFee = 10;
        return price + tax + shippingFee;
    } else {
        return price + tax;
    }
}
```
Clustering based on operations

Abstract type inference [ISSTA 2006]

```java
int totalCost(int miles, int price, int tax) {
    int year = 2016;
    if ((miles > 1000) && (year > 2000)) {
        int shippingFee = 10;
        return price + tax + shippingFee;
    } else {
        return price + tax;
    }
}
```
Clustering based on variable names

Compute variable name similarity for var$_1$ and var$_2$

1. **Tokenize** each variable into dictionary words
   - in_authskey15 $\Rightarrow$ \{“in”, “authentications”, “key”\}
   - Expand abbreviations, best-effort tokenization

2. **Compute** word similarity
   - For all $w_1 \in \text{var}_1$ and $w_2 \in \text{var}_2$, use WordNet (or edit distance)

3. **Combine** word similarity into variable name similarity
   - $\text{maxwordsim}(w_1, \text{var}_2) = \max_{w_2 \in \text{var}_2} \text{wordsim}(w_1, w_2)$
   - $\text{varsim}(\text{var}_1, \text{var}_2) = \text{average} \ \max_{w_1 \in \text{var}_1} \text{maxwordsim}(w_1, \text{var}_2)$
Results

• Ran on grep and Exim mail server
• Top-ranked mismatch indicates an undesired variable interaction in grep
  
  `if (depth < delta[tree->label])`
  
  `delta[tree->label] = depth;`

• Loses top 3 bytes of depth
• Not exploitable because of guards elsewhere in program, but not obvious here
Related work

• Reusing identifier names is error-prone [Lawrie 2007, Deissenboeck 2010, Arnaoudova 2010]
• Identifier naming conventions [Simonyi]
• Units of measure [Ada, F#, etc.]
• Tokenization of variable names [Lawrie 2010, Guerrouj 2012]
## Applying NLP to software engineering

### Problems
- inadequate diagnostics
- incorrect operations
- missing tests
- unimplemented functionality

### NL sources
- error messages
- variable names
- code comments
- user questions

### NLP techniques
- document similarity
- word semantics
- parse trees
- translation

[ISSTA 2016]
Test oracles (assert statements)

A test consists of
- an input (for a unit test, a sequence of calls)
- an oracle (an assert statement)

Programmer-written tests
- often trivial oracles, or too few tests

Automatic generation of tests:
- inputs are easy to generate
- oracles remain an open challenge

Goal: create test oracles from what programmers already write
Automatic test generation

- Code under test:
  ```java
  public class FilterIterator implements Iterator {
    public FilterIterator(Iterator i, Predicate p) {...}
    public Object next() {...}
    ...
  }
  ```

- Automatically generated test:
  ```java
  public void test() {
    FilterIterator i = new FilterIterator(null, null, null);
    i.next(); // Throws NullPointerException!
  }
  ```

Did the tool discover a bug?

It could be:
1. Expected behavior
2. Illegal input
3. Implementation bug
Automatically generated tests

• A test generation tool outputs:
  • Failing tests – indicates a program bug
  • Passing tests – useful for regression testing

• Without a specification, the tool guesses whether a given behavior is correct
  • False positives: report a failing test that was due to illegal inputs
  • False negatives: fail to report a failing test because it might have been due to illegal inputs
Programmers write code comments

Javadoc is standard procedure documentation

/**
 * Checks whether the comparator is now locked against further changes.
 *
 * @throws UnsupportedOperationException
 * if the comparator is locked
 */

protected void checkLocked() {...}
Javadoc comment and assertion

class MyClass {

    ArrayList allFoundSoFar = ...;

    boolean canConvert(Object arg) { ... }

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

Condition for exception: myMethod should throw iff ...

    (!allFoundSoFar.contains(element)
     && !canConvert(element))
The element is greater than the current maximum.

```
elt.compareTo(currentMax) > 0
```
Text to code: Toradocu algorithm

1. Parse `@param`, `@return`, and `@throws` expressions using the Stanford Parser
   • Parse tree, grammatical relations, cross-references
   • Challenges:
     • Often not a well-formed sentence; code snippets as nouns/verbs
     • Referents are implicit, assumes coding knowledge

2. Match each subject to a Java element
   • Pattern matching
   • Lexical similarity to identifiers, types, documentation

3. Match each predicate to a Java element

4. Create assert statement from expressions and methods
Results

Accuracy on 857 Javadoc tags:
• 97% precision
• 72% recall
Can tune parameters to favor either metric
Pre-processing and pattern-matching are important

Discovered specification errors

Improving test generation tools:
• Reduced false positive test failures in EvoSuite by $\geq 1/3$
• Also improved Randoop, but by less
Related work

Heuristics
• JCrasher, Crash’n’Check [Csallner’04, Csallner’05]
• Randoop [Pacheco’07]

Specifications
• ASTOOT [Doong’94]
• Models, contracts, ...

Properties
• Cross-checking oracles [Carzaniga’14]
• Metamorphic testing [Chen’13]
• Symmetric testing [Gotlieb’03]

Natural language documentation
• iComment, aComment, @tComment [Tan’07, Tan’11, Tan’12]
## Applying NLP to software engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>NL sources</th>
<th>NLP techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>inadequate</td>
<td>error</td>
<td>document</td>
</tr>
<tr>
<td>diagnostics</td>
<td>messages</td>
<td>similarity</td>
</tr>
<tr>
<td>incorrect</td>
<td>variable</td>
<td>word</td>
</tr>
<tr>
<td>operations</td>
<td>names</td>
<td>semantics</td>
</tr>
<tr>
<td>missing</td>
<td>code</td>
<td>parse</td>
</tr>
<tr>
<td>tests</td>
<td>comments</td>
<td>trees</td>
</tr>
<tr>
<td>unimplemented</td>
<td>user</td>
<td>translation</td>
</tr>
<tr>
<td>functionality</td>
<td>questions</td>
<td></td>
</tr>
</tbody>
</table>
Machine translation

English: “My hovercraft is full of eels.”
Spanish: “Mi aerodeslizador está lleno de anguilas.”

English: “Don’t worry.”
Spanish: “No te preocupes.”
Sequence-to-sequence recurrent neural network translators

Input, hidden, and output functions are inferred from training data using probability maximization.
Tellina: text to commands

• Training data: ~5000 ⟨text, command⟩ pairs
  • Collected manually from webpages, plus cleaning
• 17 file system utilities, > 200 flags, 9 types of constants
  • Compound commands: ( ), &&, ||
  • Nesting: |, $(), <() 
• Strings are opaque; no command interpreters (awk, sed)
• No bash compound statements (for)
Results

Accuracy for Tellina’s first output:
• Structure of command (without constants): 69%
• Full command (with constants): 30%

User experiment:
• Tellina makes users 22% more efficient
  • Even though it rarely gives a perfect command
• Qualitative feedback
  • Most participants wanted to continue using Tellina (5.8/7 Likert scale)
  • Partially-correct answers were helpful, not too hard to correct
  • Output bash commands are sometimes non-syntactic or subtly wrong
  • Needs explanation of meaning of output bash commands
Related work

Neural machine translation
  • Sequence-to-sequence learning with neural nets [Sutskever 2014]
  • Attention mechanism [Luong 2015]

Semantic parsing
  • Translating natural language to a formal representation [Zettlemoyer 2007, Pasupat 2016]

Translating natural language to DSLs
  • If-this-then-that recipes [Quirk 2015]
  • Regular expressions [Locascio 2016]
  • Text editing, flight queries [Desai 2016]
Other software engineering projects

• Analyzing programs before they are written
• Gamification (crowd-sourcing) of verification
• Evaluating and improving fault localization
• Pluggable type-checking for error prevention

• ... many more: systems, synthesis, verification, etc.

UW is hiring! Faculty, postdocs, grad students
<table>
<thead>
<tr>
<th>Problems</th>
<th>NL sources</th>
<th>NLP techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>inadequate diagnostics</td>
<td>error messages</td>
<td>document similarity</td>
</tr>
<tr>
<td>incorrect operations</td>
<td>variable names</td>
<td>word semantics</td>
</tr>
<tr>
<td>missing tests</td>
<td>code comments</td>
<td>parse trees</td>
</tr>
<tr>
<td>unimplemented functionality</td>
<td>user questions</td>
<td>translation</td>
</tr>
</tbody>
</table>
Analyzing text

• iComment [Tan 2007]: pattern matching for null
• N-gram models: code completion [Hindle 2011], predict variable names, whitespace [Allemanis 2014]
• Mining variable names [Pollock et al.]
• Code → comments [Sridhara 2010]
• DARPA Big Mechanism (read cancer papers)
• JSNice [Raychev 2015]: learn rules for identifiers and types
Analyzing other artifacts by machine learning over the program

• Tests (dynamic invariant detection)
• Mining software repositories
• Defect prediction
• Code completion
• Clone detection
• ... many, many more
Machine learning + software engineering

• Software is more than source code
• Formal program analysis is useful, but insufficient
• Analyze and generate all software artifacts

A rich space for further exploration