Natural language processing meets software testing

Michael Ernst

UW CSE

Joint work with Juan Caballero, Alberto Goffi, Alessandra Gorla, Mauro Pezze, Irfan Ul Haq, and Sai Zhang

March 15, 2016

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

$$
\begin{array}{llll}
\mathtt{h} & \in & \mathtt{Heap} & = \mathtt{Addr} \rightarrow \mathtt{Obj} \\
\iota & \in & \mathtt{Addr} & = \text{Set of Addresses} \cup \{\mathtt{null}_a\} \\
\mathtt{o} & \in & \mathtt{Obj} & = {}^{r}\mathtt{Type},\ \mathtt{Fields} \\
{}^{r}\mathtt{T} & \in & {}^{r}\mathtt{Type} & = \mathtt{OwnerAddr}\ \mathtt{ClassId}\texttt{<}\overline{{}^{r}\mathtt{Type}}\texttt{>} \\
\mathtt{Fs} & \in & \mathtt{Fields} & = \mathtt{FieldId} \rightarrow \mathtt{Addr} \\
\iota & \in & \mathtt{OwnerAddr} & = \mathtt{Addr} \cup \{\mathtt{any}_a\} \\
{}^{r}\Gamma & \in & {}^{r}\mathtt{Env} & = \overline{\mathtt{TVarId}\ {}^{r}\mathtt{Type}};\ \overline{\mathtt{ParId}\ \mathtt{Addr}}
\end{array}
$$

$$
\begin{array}{llll}
\mathtt{P} & \in & \mathtt{Program} & ::= \overline{\mathtt{Class}},\ \mathtt{ClassId},\ \mathtt{Expr} \\
\mathtt{Cls} & \in & \mathtt{Class} & ::= \mathtt{class\ ClassId}\texttt{<}\overline{\mathtt{TVarI}} \\
 & & & \quad \mathtt{extends\ ClassId}\texttt{<}\overline{{}^{s}\mathtt{Ty}} \\
 & & & \quad \{\ \overline{\mathtt{FieldId}\ {}^{s}\mathtt{Type};}\ \mathtt{Met} \\
{}^{s}\mathtt{T} & \in & {}^{s}\mathtt{Type} & ::= {}^{s}\mathtt{NType}\ \mid\ \mathtt{TVarId} \\
{}^{s}\mathtt{N} & \in & {}^{s}\mathtt{NType} & ::= \mathtt{OM\ ClassId}\texttt{<}\overline{{}^{s}\mathtt{Type}}\texttt{>} \\
\mathtt{u} & \in & \mathtt{OM} & ::= \\
\mathtt{mt} & \in & \mathtt{Meth} & ::= \\
 & & \mathtt{MethSig} & ::= \\
\mathtt{w} & \in & \mathtt{Purity} & ::= \\
\mathtt{e} & \in & \mathtt{Expr} & ::= \\
 & & & \quad \mathtt{Expr.MethId}\texttt{<}{}^{s}\mathtt{Type}\texttt{>}(\mathtt{Expr})\ \mid \\
 & & & \quad \mathtt{new}\ {}^{s}\mathtt{Type}\ \mid\ ({}^{s}\mathtt{Type})\ \mathtt{Expr} \\
{}^{s}\Gamma & \in & {}^{s}\mathtt{Env} & ::= \overline{\mathtt{TVarId}\ {}^{s}\mathtt{NType}};\ \overline{\mathtt{ParId}\ {}^{s}\mathtt{Type}}
\end{array}
$$

$$
\text{OS-Upd}\ \frac{\begin{array}{c}\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \mathtt{h}_0, {}^{r}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \\ \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]\end{array}}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 \rightsquigarrow \mathtt{h}',}
$$

$$
\text{OS-Read}\ \frac{\begin{array}{c}\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \iota = \mathtt{h}'(\iota_0)\downarrow_2(\mathtt{f})\end{array}}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota}
$$

$$
\text{GT-Upd}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \mathtt{u}_0\ \mathtt{C}_0\texttt{<\_>} \\ \mathtt{T}_1 = \mathit{fType}(\mathtt{C}_0, \mathtt{f}) \\ \Gamma \vdash \mathtt{e}_2 : \mathtt{N}_0 \rhd \mathtt{T}_1 \\ \mathtt{u}_0 \neq \mathtt{any} \qquad \mathit{rp}(\mathtt{u}_0, \mathtt{T}_1)\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 : \mathtt{N}_0 \rhd \mathtt{T}_1}
$$

$$
\text{GT-Read}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \_\ (}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{N}_0 \rhd \mathit{fType}(\mathtt{C}_0, \mathtt{f})}
$$

$$
\left.\begin{array}{l}
\mathtt{h} \vdash {}^{r}\Gamma : {}^{s}\Gamma \\
\mathtt{h} \vdash \iota_1 : \mathit{dyn}({}^{s}\mathtt{N}, \mathtt{h}, {}^{r}\ldots) \\
\mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{s}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1) \\
{}^{s}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\texttt{<>} \\
\mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{r}\Gamma(\mathtt{this}) \\
\mathit{free}({}^{s}\mathtt{T}) \subseteq \mathit{dom}(\mathtt{C}_N)
\end{array}\right\} \implies \mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma)
$$

$$
\text{DYN}\ \frac{\begin{array}{c}{}^{r}\mathtt{T} = \iota'\ \_\texttt{<\_>} \qquad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\texttt{<:}\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>} \qquad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\texttt{<:}\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}_a}\texttt{>} \Rightarrow \iota \vdash \overline{{}^{r}\mathtt{T}}\ {}^{r}\texttt{<:}\ \overline{{}^{r}\mathtt{T}_a} \\ \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathrm{free}({}^{s}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}\end{array}}{\mathit{dyn}({}^{s}\mathtt{T}, \iota, {}^{r}\mathtt{T}, (\overline{\mathtt{X}'\ {}^{r}\mathtt{T}'};\_)) = {}^{s}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}\mathtt{T}/\mathtt{X}}, \overline{{}^{r}\mathtt{T}'/\mathtt{X}'}]}
$$

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?

Analysis of a natural object

- Machine learning over executions
- Version control history analysis
- Bug prediction
- Upgrade safety
- Prioritizing warnings
- Program repair

Natural language in programs

This talk:

1. Variable names:
   find undesired variable interactions
2. Error messages and user manuals:
   find inadequate diagnostic messages
3. Procedure documentation:
   generate test oracles

Undesired variable interactions

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

totalPrice = itemPrice + shippingDistance;
```

Undesired variable interactions

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

Differences indicate bugs or poor variable names

Clustering based on operations

Abstract type inference [ISSTA 2006]

```
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]

```
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

1. Tokenize each variable into dictionary words
   - **`in_authskey15`** ⇒ {"in", "authentications", "key"}
   - Expand abbreviations, best-effort tokenization
2. Compute word similarity
   - For all w1 ∈ var1 and w2 ∈ var2, use WordNet or edit distance
3. Combine word similarity into variable name similarity
   - maxwordsim(w1) = maximum wordsim(w1, w2) for w2 ∈ var2
   - varsim(var1) = average maxwordsim(w1) for w1 ∈ var1

Results

- Found 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

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 …"
- Better: "`--port_num` should be an integer"

Goal: detect such problems before shipping the code

Challenges for proactive detection of inadequate diagnostic messages

- How to *trigger a configuration error*?

- How to *determine the inadequacy* of a diagnostic message?

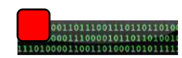
ConfDiagDetector's solutions

- How to *trigger a configuration error*?
  - Configuration mutation + run system tests

configuration

+

system tests



failed tests ≈ triggered errors

- How to *determine the inadequacy* of a diagnostic message?
  - Use a NLP technique to check its semantic meaning

Similar semantic meanings?

Diagnostic messages output by failed tests

User manual

Message analysis

- A message is adequate, if it
  - contains the mutated option name or value

  OR

  - has a similar semantic meaning with the manual description

Text similarity technique [Mihalcea’06]

A message

Manual description

Has similar semantic meanings, if many words in them have similar meanings

<u>Example:</u>

~~The~~ program goes wrong

~~The~~ software fails

- *Remove all stop words*
- *For each word in the diagnostic message, tries to find the similar words in the manual*
- *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
  - 2% false positive rate

Test oracles for exceptional behavior

Exceptional behavior is a significant source of failures, but is under-tested (significantly less coverage)

Goal: create test oracles (= `assert` statements)

Although programmers may not write tests, the programmer does provide other indications: procedure documentation (e.g., Javadoc)

```
/**
 * Checks whether the comparator is now locked
 * against further changes.
 *
 * @throws UnsupportedOperationException if the
 * comparator is locked
 */
protected void checkLocked() {...}
```

Text to code

1. Parse the @throws expression 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
    - Semantic similarity
    - Lexical similarity to identifiers, types, documentation
3. Match each predicate to a Java element
4. Create assert statement from expressions and methods

Automatically generated tests

- A test generation tool outputs:
  - Passing tests – useful for regression testing
  - Failing tests – indicates a program bug
- Without a formal specification, 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
- Results: Reduced false positive test failures in EvoSuite by 1/3 or 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