Automated Program Repair & Verification

Coming up

- This Thursday, Beta presentations!
- Next week:
  - Tuesday, Lecture on ethics in software engineering
  - Thursday, no class — at home, online assignment/activity
    - Graded — part of the participation grade (3% of overall class grade)
    - Can optionally opt into sharing anonymized data with researchers
  - Also Thursday, Beta due!

The Cost of Poor Software Quality in the US

Software engineers spend 35-50% of their time validating and debugging software.

Cost of debugging, testing, and verification accounts for 50-75% of the software development budgets.


Software engineers spend 35-50% of their time validating and debugging software.

Cost of debugging, testing, and verification accounts for 50-75% of the software development budgets.


Machine Learning in Software Engineering

A simpler problem: Automated program repair

Source code $\rightarrow$ patched program $\rightarrow$ test suite

A simpler problem: Automated program repair

Source code $\rightarrow$ patched program $\rightarrow$ test suite
Program repair techniques

- Tweak the program
- Check if tests pass
- If not, repeat

APR is a form of machine learning

- first, many techniques rely on ML to learn
  - where to edit the code
  - how to edit the code
  - how to decide which patches are good
- second, the underlying problem is learning a function (program) using training data (tests)

How well does APR work?

- Evaluated 4 techniques
  - GenProg
  - Par
  - TrpAutoRepair
  - SimFix
- Measured patch quality
- Measured what affects patch quality

Quality vs. quantity

When applied to real-world Java code, APR produces patches for 10.6-19.0% of the defects
Quality vs. quantity

Quality vs. quantity

<table>
<thead>
<tr>
<th>Technique</th>
<th>Minimum</th>
<th>Mean</th>
<th>Median</th>
<th>Maximum</th>
<th>100% Quality Patches</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoProg</td>
<td>64.8%</td>
<td>98.7%</td>
<td>98.8%</td>
<td>100.0%</td>
<td>24.7%</td>
</tr>
<tr>
<td>Par</td>
<td>64.9%</td>
<td>98.7%</td>
<td>98.4%</td>
<td>100.0%</td>
<td>13.8%</td>
</tr>
<tr>
<td>StarFix</td>
<td>87.9%</td>
<td>96.1%</td>
<td>96.5%</td>
<td>100.0%</td>
<td>45.7%</td>
</tr>
<tr>
<td>TryAutoRepair</td>
<td>64.9%</td>
<td>96.4%</td>
<td>96.6%</td>
<td>100.0%</td>
<td>39.5%</td>
</tr>
</tbody>
</table>

Less than half (14-46%) of the patches are correct.

Does APR at least improve things a bit?

<table>
<thead>
<tr>
<th>Technique</th>
<th>Change in Quality Due to Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>GeoProg</td>
<td>-30.9%</td>
</tr>
<tr>
<td>Par</td>
<td>-31.9%</td>
</tr>
<tr>
<td>StarFix</td>
<td>-24.9%</td>
</tr>
<tr>
<td>TryAutoRepair</td>
<td>-30.9%</td>
</tr>
</tbody>
</table>

Takeaway: Tests are an imperfect oracle, so APR suffers, producing low-quality patches.

Can we find a domain with better oracles?
Formal verification allows proving software correct

Interactive theorem provers for formal verification

Industrial impact of theorem proving

Prohibitively difficult
Verified software requires a lot of time and a lot of proofs in proportion to code

Proposal: Use APR-style technology to synthesize proofs

Step 1: Build a predictive model
Step 2: Guide search with the model
Proposal: Use APR-style technology to synthesize proofs

Step 2: Guide search with the model

How to learn a predictive model

Step 1: Build a predictive model

- proof state, previous tactic
- tactic: axiom, induction, apply, etc.
- partial proof:
  - intros n; induction n;
  - simpl;
  - backtrack
- predicted next proof steps:
  - predicted tactic
  - predicted next proof steps

Tactic: LetTac  

-predicted tactic
- predicted next proof steps
- predicted next proof steps

Tactic: TacTok (OOPSLA'20)

TacTok models partial proof and the current proof state, together
CoqGym Dataset

- 123 open-source software projects in Coq
- 70,856 theorems
- Broken down into 96 projects (57,719 proofs) for training and 27 projects (13,137 theorems) for testing

https://github.com/princeton-vl/CoqGym

[Tang and Deng, Learning to Prove Theorems via Interacting with Proof Assistants, ICML'19]

TacTok vs. ASTactic vs. SeqOnly

<table>
<thead>
<tr>
<th>TacTok</th>
<th>ASTactic</th>
<th>SeqOnly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,388</td>
<td>1,302</td>
<td>1,277</td>
</tr>
<tr>
<td>180 (13.7%)</td>
<td>180 (14.1%)</td>
<td>180 (13.9%)</td>
</tr>
<tr>
<td>412 (30.4%)</td>
<td>412 (31.3%)</td>
<td>412 (30.5%)</td>
</tr>
<tr>
<td>84 (6.2%)</td>
<td>84 (6.3%)</td>
<td>84 (6.2%)</td>
</tr>
<tr>
<td>712 (53.2%)</td>
<td>712 (53.2%)</td>
<td>712 (53.2%)</td>
</tr>
<tr>
<td>224 (1.7%)</td>
<td>224 (1.7%)</td>
<td>224 (1.7%)</td>
</tr>
<tr>
<td>8,972 (83.2%) unproven theorems</td>
<td>8,972 (83.2%) unproven theorems</td>
<td>8,972 (83.2%) unproven theorems</td>
</tr>
</tbody>
</table>

TacTok vs. ASTactic vs. CoqHammer

<table>
<thead>
<tr>
<th>TacTok</th>
<th>ASTactic</th>
<th>CoqHammer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,388</td>
<td>1,302</td>
<td>1,322</td>
</tr>
<tr>
<td>180 (13.7%)</td>
<td>180 (14.1%)</td>
<td>180 (13.9%)</td>
</tr>
<tr>
<td>412 (30.4%)</td>
<td>412 (31.3%)</td>
<td>412 (30.5%)</td>
</tr>
<tr>
<td>84 (6.2%)</td>
<td>84 (6.3%)</td>
<td>84 (6.2%)</td>
</tr>
<tr>
<td>712 (53.2%)</td>
<td>712 (53.2%)</td>
<td>712 (53.2%)</td>
</tr>
<tr>
<td>224 (1.7%)</td>
<td>224 (1.7%)</td>
<td>224 (1.7%)</td>
</tr>
<tr>
<td>8,972 (83.2%) unproven theorems</td>
<td>8,972 (83.2%) unproven theorems</td>
<td>8,972 (83.2%) unproven theorems</td>
</tr>
</tbody>
</table>

Diva (ICSE’22)

- 2 key observations:
  - Machine learning is often noisy
  - Theorem prover serves as an oracle to turn that noise into signal.

https://github.com/LASER-UMASS/Diva/

Diva vs. state-of-the-art

Diversity inherent in ML increases the proving power 68%-77% over prior search-based synthesis tools, and 27% over CoqHammer.

https://github.com/LASER-UMASS/Diva/
Prior work: transformers for Isabelle

- **Context**
  - **Proof state**
  - **LLM**

**JO3M**

- **Proof state**
  - **Disproof**
  - **Proof state**

**Proves 39% of PISA benchmark**

**Deduction**

**Baldur: whole proof generation & repair**

Evaluate

- **Baldur** whole-proof generation & repair

**PISA test set success rate**

- **LLM** + **Sledgehammer**
  - **Thor** (LLM + Sledgehammer)

- **Baldur (8b generate with context)**
  - **Thor**

Baldur + Thor prove nearly 2/3 of PISA test set!

Generate vs Generate+Repair

- Generate+Repair outperforms Generate only

Generate+Repair with no error message

- Error message is crucial for repair approach
Generate with context

Proof context helps improve proof generation.

Fully Automated Formal Verification

Machine learning and meta-heuristic search can fully automate some bug-repair and formal verification.

While APR underperforms because it is driven by an unreliable oracle, formal verification is a killer app for APR because the theorem prover provides a reliable oracle.