Testing Evolving Software
Current Practice and Future Promise

Mary Jean Harrold
ADVANCE Professor of Computing
College of Computing
Georgia Institute of Technology

Problems for Evolving Software

- Estimating evolution (maintenance) costs
- Updating requirements, design, code, documentation
- Predicting faulty parts of modified system
- Managing software repositories
Problems for Evolving Software

- Estimating evolution (maintenance) costs
- Updating requirements, design, code, documentation
- Predicting faulty parts of modified system
- Testing modified software
- Regression testing
- Managing software repositories

Testing Evolving Software

- Interest
- Problems
  - Achievements
  - Challenges
- Industry—academic collaboration
Collaboration With Industry

Common Problem
- Changes require rapid modification and testing for quick release
- Causing released software to have many defects

Research Question
How can we test well to gain confidence in the changes in an efficient way before release of changed software?

Approach
- Concentrate testing around the changes
- Automate the regression testing process

Testing Evolving Software

Program P

Develop T for P

Augment T for untested adequacy requirements

Modify P

Select subset of T to rerun

Execute

Identify faults

Assess outcome

Assess adequacy

Monitor to improve quality

Release P

T

F

T
Testing Evolving Software

- Present problem
- Overview current status, achievements
- Discuss challenges and open issues
Select Subset of T to Rerun

Assess adequacy

Augment T for untested adequacy requirements

Program P

P’ Version of P

T
Select Subset of T to Rerun

Which test cases in T should be rerun to test P’?

P’ Version of P

Solution
Partition T into two subsets
• run one on P’
• don’t run the other
Select Subset of T to Rerun

1. Construct representation $G$ for $P$

2. Associate test cases in $T$ with entities in $G$

3. Build $G'$ and compare $G$ and $G'$ to find dangerous entities

4. Select test cases based on dangerous entities
**Empirical Studies**

**Goal:** To determine savings in execution time

**Subjects**

<table>
<thead>
<tr>
<th>Program Type</th>
<th>Versions</th>
<th>Procedures</th>
<th>KLOC</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Program</td>
<td>Empire</td>
<td>5</td>
<td>766</td>
<td>50</td>
</tr>
<tr>
<td>Java Programs</td>
<td>Daikon</td>
<td>5</td>
<td>824</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td>JBoss</td>
<td>5</td>
<td>2,403</td>
<td>~1K</td>
</tr>
</tbody>
</table>

**Procedure**

For each pair of versions $v_i$, $v_{i+1}$, measure:

- Time to re-run $v_{i+1}$ on all test cases in $T$
- Time to select $T'$ + Time to run $T'$ on $v_{i+1}$

Compare times:

- Save if $B < A$

---

**Savings in Testing Time Using DejaVOO**

<table>
<thead>
<tr>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daikon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jboss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Savings in Testing Time Using DejaVoo

<table>
<thead>
<tr>
<th>Version</th>
<th>Empire</th>
<th>Daikon</th>
<th>Jboss</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>93%</td>
<td>36%</td>
<td>63%</td>
</tr>
<tr>
<td>V3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if T’ has too many test cases for allotted time? What if want to run most important test cases in T’ first?
Select Subset of T to Rerun

What if
- T’ has too many test cases for allotted time?
- want to run most important test cases in T’ first?

Solution
Order (prioritize) T’
- find faults earlier
- get coverage earlier
- etc.

Achievements: Research

- Application to different models of the system
- Empirical evidence of effectiveness on different kinds of programs written in various languages
- Evidence of the effectiveness of techniques that use simple program information
Achievements: Commercialization

Google Testar
• Selective testing tool for Java
• Works with JUnit
• Records coverage data about JUnit tests by instrumenting bytecode
• Computes, stores, and compares checksums to identify changes
• Also computes and reports coverage of methods

Challenges

Good selection/prioritization techniques
• Regression testing at the system level
• Systems with nondeterministic behavior
• Systems that are developed by distributed teams

Transfer of techniques to industry
• Automation of regression testing
• Gathering information required for selection and prioritization
• Integrating into existing testing toolsets being used in industry
Assess Adequacy

Select subset of T to rerun

Augment T for untested adequacy requirements

Assess Adequacy

How well do T, T’, T’’ or any test suites exercise P’ with respect to changes?

Do the test cases exercise the changes so that they will affect execution?
Program and Modified Version

Procedure Avg
S1  count = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4    if (n<0)
S5      return(n)
else
S6      nums[count] = n
S7      count++
endif
S8    fread(fptr,n)
endwhile
S9  avg = mean(nums,count)
S10 return(avg)

Procedure Avg'
S1' count = 0
S2' fread(fptr,n)
S3' while (not EOF) do
S4'   if (n<=0)
S5'      return(n)
else
S6'      nums[count] = n
S7'      count++
endif
S8' fread(fptr,n)
endwhile
S9' avg = mean(nums,count)
S10' return(avg)

Criteria for change-impact propagation
• Execution of the change

DejaVOO (and other criteria for changes) require execution of the change
Program and Modified Version

Criteria for change-impact propagation

- **Execution of the change**
- **Infection of the state after change**
- **Propagation of state to output where it can be observed**

Program and Modified Version

Our new technique aims to add these requirements to the change criteria
Compute Change Testing Requirements

Procedure Avg
S1  count = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4   if (n<0) \[ \text{if } (n<=0) \]
S5     return(n)
S6   else
S7     nums[count] = n
S8     count++
S9    fread(fptr,n)
S10   endwhile
S9  avg = mean(nums,count)
S10 return(avg)

S4

Infection: Path
condition in Avg after
S4 and path condition
in Avg’ after S4’ differ

Condition for infection:
(n<=0) and not (n<0)
\rightarrow n must be 0 after S4’
Compute Change Testing Requirements

Procedure Avg
S1  count = 0
S2  fread(fpfrt,n)
S3  while (not EOF) do
S4    if (n<0)  \[\text{if } (n\leq 0)\]
S5      return(n)
S6      nums[count] = n
S7      count++
S8    fread(fpfrt,n)
S9  avg = mean(nums,count)
S10 return(avg)

S7
Infection: Value of state after execution of S7 in Avg and S7’ in Avg’ must differ

Condition for infection:

After
• S7 in Avg, count=count+1
• corresponding location in Avg’, count=count
→ count̸=count+1,
→ any value of count

Infection

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(n)</th>
<th>PC’</th>
<th>SS’(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>N</td>
<td>true</td>
<td>N_0</td>
</tr>
</tbody>
</table>

PC—path condition
SS—symbolic state
Perform symbolic execution from before change to get conditions
**Compute Change Testing Requirements**

**Procedure Avg**

1. `count = 0`
2. `fread(fp1tr,n)`
3. `while (not EOF) do`
   - `if (n<0)`
     - `return(n)`
   - `else`
     - `nums[count] = n`
     - `count++`
   - `endif`
4. `fread(fp1tr,n)`
5. `while (not EOF) do`
6. `avg = mean(nums,count)`
7. `return(avg)`

**Infection**

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>N₀</td>
</tr>
<tr>
<td>(N₀&lt;0)</td>
<td>N₀</td>
</tr>
<tr>
<td>(N₀&gt;0)</td>
<td>N₀</td>
</tr>
</tbody>
</table>

**Propagation**

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>N₀</td>
</tr>
<tr>
<td>(N₀&lt;0)</td>
<td>N₀</td>
</tr>
<tr>
<td>(N₀&gt;0)</td>
<td>N₀</td>
</tr>
</tbody>
</table>

**PC—path condition**

**SS—symbolic state**

Perform symbolic execution from before change to get conditions.
Compute Change Testing Requirements

Procedure Avg
S1  count = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4    if (n<0) if (n<0)
S5      return(n)
S6    else
S7      nums[count] = n
S8      count++
S9    endwhile
S10  avg = mean(nums,count)
S11  return(avg)

Propagation

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(n)</th>
<th>PC'</th>
<th>SS'(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>true</td>
<td>PC'</td>
<td>true</td>
</tr>
<tr>
<td>SS(n)</td>
<td>N_0</td>
<td>SS'(n)</td>
<td>N_0</td>
</tr>
<tr>
<td>(N_0&lt;0) or (N_0&gt;0)</td>
<td>N_0</td>
<td>(N_0&lt;0) or (N_0&gt;0)</td>
<td>N_0</td>
</tr>
<tr>
<td>temp = 5/N_0</td>
<td>temp = 5/N_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute Change Testing Requirements

Procedure Avg
S1  count = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4    if (n<0) if (n<0)
S5      return(n)
S6    else
S7      nums[count] = n
S8      count++
S9    endwhile
S10  avg = mean(nums,count)
S11  return(avg)

Propagation

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(n)</th>
<th>PC'</th>
<th>SS'(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>true</td>
<td>PC'</td>
<td>true</td>
</tr>
<tr>
<td>SS(n)</td>
<td>N_0</td>
<td>SS'(n)</td>
<td>N_0</td>
</tr>
<tr>
<td>(N_0&lt;0) or (N_0&gt;0)</td>
<td>N_0</td>
<td>(N_0&lt;0) or (N_0&gt;0)</td>
<td>N_0</td>
</tr>
<tr>
<td>temp = 5/N_0</td>
<td>temp = 5/N_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Compute Change Testing Requirements**

### Procedure Avg

1. \(\text{count} = 0\)
2. \(\text{fread(fptr,n)}\)
3. while (not EOF) do
   4. if \((n<0)\) then \(\text{return}(n)\)
   5. else
      6. \(\text{nums}[\text{count}] = n\)
      7. \(\text{count}++\)
   8. \(\text{fread(fptr,n)}\)
   9. endwhile
10. \(\text{avg} = \text{mean}(\text{nums},\text{count})\)
11. return(\(\text{avg}\))

### Propagation

<table>
<thead>
<tr>
<th></th>
<th>Avg</th>
<th>Avg'</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>SS(n)</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>(N_0)</td>
<td>true</td>
</tr>
<tr>
<td>(N_0&lt;0) \ or \ (N_0&gt;=0)</td>
<td>(N_0)</td>
<td>(N_0) \ or \ (N_0&gt;0)</td>
</tr>
<tr>
<td>(N_0&gt;=0)</td>
<td>(\text{temp} = 5/N_0)</td>
<td>(N_0)</td>
</tr>
<tr>
<td>(N_0&gt;0)</td>
<td>(N_0)</td>
<td>(N_0)</td>
</tr>
<tr>
<td>and so on</td>
<td></td>
<td>and so on</td>
</tr>
</tbody>
</table>

### But

- Symbolic execution on the entire program is expensive.
- May not scale to large programs.
- Etc.

Our technique has two ways to improve efficiency.

12. \(\text{avg} = \text{mean}(\text{nums},\text{count})\)
13. return(\(\text{avg}\))
Compute Change Testing Requirements

1. Perform partial symbolic execution (PSE) beginning immediately before the change
   - computes conditions in terms of variables immediately before change
   - avoids symbolic execution from beginning of program to change
   Don’t need to solve conditions—can still monitor for their satisfaction

| S8 | fread(fptr,n)       | (N_o>0)  | N_o   | (N_o<0)  |
|    | endwhile           | and so on|       | and so on |
| S9 | avg = mean(nums,count) |
| S10| return(avg)        |

2. Perform PSE for some specified distance (user selected) instead of to output statements
   - computes conditions on states at intermediate points (i.e., distances)
   - bounds depth using slicing-like dependences, avoids symbolic execution to outputs
   Greater distances improve confidence in propagation to output

| S9 | avg = mean(nums,count) |
| S10| return(avg)            |
Compute Change Testing Requirements

1. PSE—conditions in terms of variables at point before change

Compute Change Testing Requirements

1. PSE—conditions in terms of variables at point before change

relevant variables
1. PSE—conditions in terms of variables at point before change
2. Distance is N data and control dependences from change (slicing like)

Distance is 0

Distance is 1
Compute Change Testing Requirements

1. PSE—conditions in terms of variables at point before change
2. Distance is N data and control dependences from change (slicing like)

Insert Probes to Record Coverage

To record adequacy (coverage of conditions)

Instrument modified program so that probes check for satisfaction of condition before change (e.g., before S4’ and before S7’)

```
Procedure Avg'
S1’ count = 0
S2’ fread(fptr,n)
S3’ while (not EOF) do
S4’ if (n<=0)
S5’ return(n)
else
S6’ nums[count] = n
S7’ endif
S8’ fread(fptr,n)
endwhile
S9’ avg = mean(nums,count)
S10’ return(avg)
```
Empirical Study

**Goal:** To compare the effectiveness of our changed-based criterion with statement-based criterion

**Subjects:** Tcas (4 versions) and Schedule (3 versions) (each version has one fault)

**Procedure:**
- Randomly generated 50 test suites per criterion
- Recorded the number of test suites that produce different outputs

Criteria Effectiveness Using MaTRIX

<table>
<thead>
<tr>
<th>Percentage revealing different behaviors</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>stmt</td>
<td>change-based</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schedule</td>
<td>stmt</td>
<td>change-based</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Achievements: Research

- New change test criteria
  - show promise for assessing adequacy of test suite around changes
  - can be performed on large programs since only area around change being evaluated
  - empirical evaluation encouraging

Challenges

Computing change impact criteria still in very early stages
- Handling multiple and interacting changes
- Implementing PSE efficiently
- Determining good distance (empirically)
- Tracking impact dynamically to check for propagation to end of program
- Creating strategy to handle false positives
- Others??
Augment T for … Requirements

Select subset of T to rerun
Assess adequacy

How can we get test cases to satisfy unsatisfied conditions?

P’ Version of P

T-T’
T’
T’
T”

T
Augment T for … Requirements

• Unsatisfied conditions can be used by developers to create new test cases
• Automatically generate test cases to satisfy conditions
  • Use of regression test suite to generate new inputs to satisfy change test requirements
  • Dynamic information provides concrete values to guide symbolic execution
  • Extension of existing work in concolic test data generation

Challenges

• Generating test cases to satisfy conditions
  • may use regression test suite and apply existing techniques
  • ensure that the techniques are efficient for large programs
Testing Evolving Software

Program P → Develop T for P → Augment T for untested adequacy requirements

Modify P → Select subset of T to rerun → Execute

Identify faults → Assess outcome → Assess adequacy

Monitor to improve quality → Release P

Industry-Academic Collaboration

Collaboration with:
- Data General (EMC)
- Microsoft
- Boeing Commercial Airplanes
- Tata Consultancy Services, Ltd. (TCS)

Top 7 lessons learned
Top 7 Lessons Learned

1. To solve real problems, need to interact with industry on a regular basis
2. To incorporate new testing technology often requires extensive change in process, so difficult to achieve
3. To show effectiveness, techniques must be evaluated on real systems
4. To evaluate, prototype must integrate into industrial environment

Top 7 Lessons Learned

5. To get developers’ help in evaluation, prototype must be usable for them
6. To get the evaluation done, need internal champion and internal developer needed for experimentation
7. To discover important problems, many ideas for interesting research emerge from collaborations
Summary

- **Regression testing approach**
  - Concentrate testing around the changes
  - Automate as much as possible
- **Components**
  - Regression test selection and prioritization
  - Test-suite adequacy assessment
  - Test-suite augmentation to satisfy adequacy requirements

Promise for Testing Evolving Software

We’re making progress on automating the regression testing process

By continuing to work on these and related problems, and collaborating with industry, we will be able to automate regression testing

Thereby providing a way to test well around changes efficiently, provide confidence in the software, and eliminate many defects
Questions?