Software Verification: An Evolution-Centric Perspective

Gregg Rothermel
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln

Supported by the National Science Foundation,
Microsoft, Lockheed Martin,
and Boeing Commercial Aircraft Group
Evolving Software

Palette CAD 1.0
- 1.0 Apollo
  - 1.01 Patches
    - 1.1 Features

Palette CAD 2.0
- 2.0 Apollo
  - 2.01 Patches
    - 2.1 Features
An Evolution-Centric Perspective on Software Verification

- Focus on evolution first
- Harness evolution
- Design for incremental validation
Overview of Presentation

• Evolving software

• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation

• Regression model checking
  – Algorithm
  – Empirical evaluation

• Ongoing and Future Work
Overview of Presentation

• Evolving software

• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation

• Regression model checking
  – Algorithm
  – Empirical evaluation

• Ongoing and Future Work
Evolving Software: A Basic Process Model

Preliminary period  Critical period

Maintenance  V & V

Preliminary period  Critical period
Evaluating Evolving Software with Regression Testing

- $P \rightarrow P'$
- $T \rightarrow T$
- $t_1$ maintenance
- $t_2$ V & V
- $t_3$
Overview of Presentation

• Evolving software
• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation
• Regression Model Checking
  – Algorithm
  – Empirical evaluation
• Ongoing and Future Work
Regression Test Selection

\[ P \quad \rightarrow \quad P' \quad \rightarrow \quad T' \quad \rightarrow \quad T' \quad \rightarrow \quad P' \quad \rightarrow \quad P \]

\[ t1 \quad \text{maintenance} \quad t2 \quad V \& V \quad t3 \]
**Control Flow Graphs**

**Procedure Avg**

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

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

<table>
<thead>
<tr>
<th>test</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>empty file</td>
<td>0</td>
</tr>
</tbody>
</table>
Procedure Avg
S1  count = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4   if (n<0)
S5     return(error)
S6   else
S7     nums[count] = n
S8   endif
S9  endwhile
S10 avg = mean(nums,count)
     return(avg)
Test History Information

<table>
<thead>
<tr>
<th>test</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>empty file</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>error</td>
</tr>
<tr>
<td>t3</td>
<td>1 2 3</td>
<td>2</td>
</tr>
</tbody>
</table>

Graph representation:
Program and Modified Version

**Procedure Avg**

S1  count = 0  
S2  fread(fptr,n)  
S3  while (not EOF) do  
S4    if (n<0)  
S5      return(error)  
S6    else  
S7      nums[count] = n  
S8    endif  
S9    fread(fptr,n)  
endwhile  
S10  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)  
S5a      print(“input error”)  
S5’    else  
S6’      return(error)  
S7’    endif  
S8’  fread(fptr,n)  
endwhile  
S9’  avg = mean(nums,count)  
S10’ return(avg)
CFG and Modified CFG

t1,t2,t3

t2,t3
Example 1

T’ = \{t2, t3\}
Example 2

$T' = \{t2, t3\}$
Example 3

$T' = \{t2\}$
**Algorithm Dejavu**

Input: P, P', T   Output: T'
1. Build CFGs G and G' for P and P'
2. Compare(G.EntryNode,G'.EntryNode)

3. Compare(N,N')
4.  mark N “N’-visited”
5.  for each pair of successors C and C’ of N and N’
6.    on equivalently labeled edges do
7.      if C is not marked “C’-visited”
8.        if C and C’ are not lexically identical
10. else
11.    Compare(C,C’)

Interprocedural Methodologies

1. Compare all pairs of procedures
2. Create & walk interprocedural representation
3. Compare all pairs of procedures identified by configuration management system
Overview of Presentation

• Evolving software
• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation
• Regression model checking
  – Algorithm
  – Empirical evaluation
• Ongoing and Future Work
Algorithm Efficiency

CFG construction: linear in program size
Graph walk (graph sizes n, n’; test set size t):

\[ O( t \times n \times n' ) \]
(with multiply-visited nodes)

\[ O( t \times \min(n,n') ) \]
(with no multiply-visited nodes)
Precision and Safety

Conditions:

1. P was correct for all tests in T
2. T contains no obsolete tests
3. Controlled regression testing
Regression Test Selection System

- Program analysis tools
  - $P, P'$

- Code instrumenter
  - $P, P', \text{cfgs}
  - $P, \text{cfg}$

- Analysis database
  - $\text{cfgs}$

- Dejavu tool
  - $\text{selected tests}$

- Test history builder
  - $\text{traces}$
  - $\text{test history}$

- Execute function $P^\wedge$
  - $P^\wedge$
  - $\text{test}$
  - $\text{trace}$
  - $\text{test history}$

Test database
  - $\text{test}$
  - $\text{trace}$

Test history
  - $\text{test history}$
Study 1: Empire

<table>
<thead>
<tr>
<th>Program</th>
<th>Procs</th>
<th>LOC</th>
<th>Vers</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>766</td>
<td>49316</td>
<td>5</td>
<td>1033</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Functions Modified</th>
<th>LOC Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>726</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>62</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>221</td>
</tr>
</tbody>
</table>
Study 1:
Test Selection Percentages

% Tests Selected

Version Number
Study 1: Cost Effectiveness

![Bar chart showing time (in hours) for each version number (1 to 5). The chart compares Retest All and Dejavu.](chart.png)
## Study 2: Windows NT Calculator

<table>
<thead>
<tr>
<th>Program</th>
<th>Funcs</th>
<th>LOCs</th>
<th>Vers</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculator</td>
<td>27</td>
<td>2145</td>
<td>9</td>
<td>3/388</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Functions Modified</th>
<th>LOCs Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>264</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>245</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>44</td>
</tr>
</tbody>
</table>
Study 2: Test Selection Percentages

Version Number

% Tests Selected

3-test
388-test
Overview of Presentation

• Evolving software

• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation

• Regression model checking
  – Algorithm
  – Empirical evaluation

• Ongoing and Future Work
Model Checking

Finite-state model of program

□ (Φ → ♦ Ω)

Property Specification (e.g., temporal logic formula)

Model Checker

OK

or

Error trace

Line 5: ...
Line 12: ...
Line 15: ...
Line 21: ...
Line 25: ...
Line 27: ...
... Line 41: ...
Line 47: ...
Basic DFS Approach

basicDFS()
1 seen := \{s_0\}
2 push(stack, s_0)
3 DFS(s_0)
end basicDFS()

DFS(s)
4 for \(\alpha \in enabled(s)\) do
5 \(s' := \alpha(s)\)
6 if error(s') then
7 ce := stack
8 exit
9 if \(s' \not\in seen\) then
10 seen := seen \cup \{s'\}
11 push(stack, s')
12 DFS(s')
13 pop(stack)
14 endfor
end DFS()
Why use Model Checking?

- Automatically check, e.g.,
  - invariants, safety & liveness properties
  - absence of dead-lock and live-lock,
  - complex event sequencing properties,

“Between the key being inserted and the key being removed, the ignition can be activated at most twice.”

- In contrast to testing, model checking gives complete coverage by exhaustively exploring all paths in a system
What Makes Model Checking Difficult to Apply in Practice?

- Model construction
- Property specification
- State explosion
- Output interpretation
What Makes Model Checking Difficult to Apply in Practice?

- Model construction
- Property specification
- State explosion
- Output interpretation
- System evolution
Regression Model Checking
Overview of Presentation

• Testing evolving software
• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation
• Regression model checking
  – Approach
  – Empirical results
• Ongoing and Future Work
Regression Model Checking (RMC) Process

1) A *recording phase* of RMC (*recordingRMC*) gathers data while model checking P in the preliminary period.

2) Dejavu computes the dangerous elements with respect to P and P’.

3) A *pruning phase* of RMC (*pruningRMC*) determines which sub-state spaces in P’ are safe and prunes them in the critical period, performing model checking only where needed.

4) RMC preserves property checking relative to a traditional model check.
RecordingRMC
RecordingRMC
Recording RMC
RecordingRMC
PruningRMC

Dangerous elements = \{1\}
RMC Implementation

Program $P$

Changes

Program $P'$

Sofya Instrumentation

Dejavu

dangerous elements

Sofya Instrumentation

$P_{instr}$

$P'_{instr}$

JPF recordingRMC

reachable elements

JPF pruningRMC
Overview of Presentation

• Evolving software
• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation
• Regression model checking
  – Algorithm
  – Empirical evaluation
• Ongoing and Future Work
Research Question

RQ1: How does RMC compare in the critical period to TMC
RQ2: How does RMC compare overall to TMC
# Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Error</th>
<th>Thd</th>
<th>Cls</th>
<th>Meth</th>
<th>LOC</th>
<th>Vers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daisy</td>
<td>AssertionViolation</td>
<td>3</td>
<td>21</td>
<td>106</td>
<td>744</td>
<td>13</td>
</tr>
<tr>
<td>Elevator</td>
<td>ArrayException</td>
<td>4</td>
<td>12</td>
<td>96</td>
<td>934</td>
<td>29</td>
</tr>
<tr>
<td>Alarmclock</td>
<td>NullPtrException</td>
<td>3</td>
<td>6</td>
<td>20</td>
<td>125</td>
<td>13</td>
</tr>
<tr>
<td>RaxExtended</td>
<td>AssertionViolation</td>
<td>6</td>
<td>11</td>
<td>23</td>
<td>127</td>
<td>9</td>
</tr>
<tr>
<td>ReplicatedWorkers</td>
<td>Deadlock</td>
<td>6</td>
<td>14</td>
<td>50</td>
<td>304</td>
<td>25</td>
</tr>
</tbody>
</table>
Variables

- Independent variables:
  - RMC and TMC (default mode of JPF)
Variables

- Independent variables:
  - RMC and TMC (default mode of JPF)
- Dependent variables:
  - execution time ($ET_{TM}, ET_{RM}$)
  - memory usage ($MU_{TM}, MU_{RM}$)
Obtaining Versions - Choices

1. Actual versions
2. Versions created by people with enough related experience to modify the program
3. Versions created by simulating changes at beginnings of methods
Threats to Validity

• External Validity
  – representativeness of object programs
  – representativeness of versions

• Internal Validity
  – faults in implementation of algorithms
  – faults in tools

• Construct Validity
  – metrics we have chosen
## Time / Memory Cost Ratios

<table>
<thead>
<tr>
<th>Program</th>
<th>TMC / RMC-Cri</th>
<th>TMC / RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daisy</td>
<td>5.0</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>1.09</td>
</tr>
<tr>
<td>Elevator</td>
<td>43.4</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>1.6</td>
<td>0.36</td>
</tr>
<tr>
<td>AlarmClock</td>
<td>1.2</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>1.14</td>
</tr>
<tr>
<td>RaxExtnded</td>
<td>79.6</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>0.75</td>
</tr>
<tr>
<td>ReplWorkers</td>
<td>43.5</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.24</td>
</tr>
</tbody>
</table>
### Time / Memory Cost Ratios

<table>
<thead>
<tr>
<th>Program</th>
<th>TMC / RMC-Cri</th>
<th>TMC / RMC-Cri</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daisy</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Elevator</td>
<td>43.4</td>
<td>1.6</td>
</tr>
<tr>
<td>AlarmClock</td>
<td>1.2</td>
<td>1.3</td>
</tr>
<tr>
<td>RaxExtended</td>
<td>79.6</td>
<td>2.1</td>
</tr>
<tr>
<td>ReplWorkers</td>
<td>43.5</td>
<td>1.1</td>
</tr>
</tbody>
</table>

RMC-Cri consistently faster than TMC; in three cases substantially

TMC uses between 1.1 and 3.0 times more memory than RMC-Cri
## Time / Memory Cost Ratios

<table>
<thead>
<tr>
<th>Program</th>
<th>TMC / RMC</th>
<th>TMC / RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daisy</td>
<td>5.0 3.0</td>
<td>0.97 1.09</td>
</tr>
<tr>
<td>Elevator</td>
<td>43.4 1.6</td>
<td>1.57 0.36</td>
</tr>
<tr>
<td>AlarmClock</td>
<td>1.2 1.3</td>
<td>0.99 1.14</td>
</tr>
<tr>
<td>RaxExtended</td>
<td>79.6 2.1</td>
<td>1.56 0.75</td>
</tr>
<tr>
<td>ReplWorkers</td>
<td>48.5 1.0</td>
<td>1.47 0.24</td>
</tr>
</tbody>
</table>

RMC is faster than TMC in 3 of 5 cases, and not much slower in 2 others.

RMC uses more memory than TMC in 3 of 5 cases, and a bit less in 2 others.
Summary

• In the critical period, RMC requires less time than TMC, and less memory
• Savings increase with costs of analysis
• Overall, RMC can also provide savings in time compared to TMC, but it does this through increased memory usage
Overview of Presentation

• Evolving software
• Regression test selection
  – Dejavu algorithm
  – Analytical and empirical evaluation
• Regression model checking
  – Algorithm
  – Empirical evaluation
• Ongoing and Future Work
Ongoing and Future Work: Regression Testing

- Test case prioritization and test augmentation
- Economic models for evaluating costs/benefits
- Configurable systems and product lines
- Regression testing real-time embedded systems
- Testing under different process models
Ongoing and Future Work: Regression Model Checking

• Reduce memory costs by using coarser coverage elements or dominator analysis

• RMC is sensitive to the structure of program states; sub-setting program states may help

• Systems move through successions of releases; RMC ignores this, a multi-version RMC approach could be more cost-effective
More Process Models

Traditional

Incremental

Continuous

maintenance

V & V
Adapting to Other Processes
An Evolution-Centric Perspective on Software Testing

- Focus on evolution first
- Harness evolution
- Design for incremental validation
Acknowledgments

• **Sponsors:**
  – National Science Foundation
  – Boeing Commercial Airplane Group
  – Microsoft
  – Lockheed Martin

• **Graduate Students:**
  – Hyunsook Do
  – Marc Fisher
  – Jung-Min Kim (U. Md.)
  – Jim Law
  – Alexey Malishevsky
  – Guowei Yang

• **Colleagues:**
  – Matthew Dwyer
  – Sebastian Elbaum
  – Mary Jean Harrold
  – Sarfraz Kurshid
  – Alex Orso
  – Adam Porter
  – David Rosenblum
Software Verification: An Evolution-Centric Perspective

Gregg Rothermel
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln

Supported by the National Science Foundation,
Microsoft, Lockheed Martin,
and Boeing Commercial Aircraft Group