Static Analysis of Mutant Subsumption
April 13, 2015

Bob Kurtz, Paul Ammann, Jeff Offutt
George Mason University
Contributions

• We developed a process to determine mutant subsumption relationships statically using symbolic execution
• We used this process to generate test cases to kill mutants for a simple example program
• We fed these tests back into dynamic analysis to remove the “unsoundness” that is characteristic of symbolic execution
• We measured the quality of the results
Motivation

• Why do we care about subsumption?
  – Mutants contain a large amount of redundancy
  – Killing one mutant often kills many others
  – Subsumption captures this redundancy and provides a useful representation

• What can we do with it once we have it?
  – Can we select a minimal set of mutants to reduce testing complexity?
“True” Subsumption

• Given a set of mutants M on artifact A, mutant $m_i$ subsumes mutant $m_j$ ($m_i \rightarrow m_j$) iff:
  – Some possible test kills $m_i$
  – All possible tests that kill $m_i$ also kill $m_j$
Dynamic Subsumption

- Dynamic subsumption approximates true subsumption using a finite test set $T$
- If $T$ contains all possible tests (it doesn’t) then dynamic subsumption $= true$ subsumption
Static Subsumption

• Static subsumption approximates true subsumption using static analysis of mutants
• If analysis is sound and complete (it isn’t) then static subsumption = true subsumption
Dominator Mutants

- Dominator mutants are not subsumed by any other mutants
- If we execute a test set that kills all the dominator mutants, then we will kill all the mutants
  - All other mutants are redundant!
To explore mutant subsumption graphs in more detail, we selected a small textbook example program

- `cal()` is a simple Java program of < 20 SLOC
  - `cal()` calculates the number of days between two dates in the same year
    - Chosen for its well-defined finite input space
- We used muJava to generate 173 mutants
Evaluating the TMSG

• We expected our various analyses to have limitations
• For comparison, we manually analyzed all 173 mutants and (with the help of a solver) determined their subsumption relationships
• The result is the true mutant subsumption graph (TMSG) for cal()
# The cal() TMSG

<table>
<thead>
<tr>
<th>TMSG</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>38</td>
</tr>
<tr>
<td>Dominator Nodes</td>
<td>10</td>
</tr>
<tr>
<td>Mutants Killed</td>
<td>145</td>
</tr>
<tr>
<td>Mutation Score</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Mutants</td>
<td>15</td>
</tr>
<tr>
<td>Dominator Precision</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Recall</td>
<td>1.00</td>
</tr>
<tr>
<td>Subsumption Precision</td>
<td>1.00</td>
</tr>
<tr>
<td>Subsumption Recall</td>
<td>1.00</td>
</tr>
<tr>
<td>Number of Tests</td>
<td>n/a</td>
</tr>
</tbody>
</table>
The cal() TMSG

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutants Killed</td>
<td>145</td>
</tr>
<tr>
<td>Mutation Score</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Mutants</td>
<td>15</td>
</tr>
<tr>
<td>Dominator Precision</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Recall</td>
<td>1.00</td>
</tr>
<tr>
<td>Subsumption Precision</td>
<td>1.00</td>
</tr>
<tr>
<td>Subsumption Recall</td>
<td>1.00</td>
</tr>
<tr>
<td>Number of Tests</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Are the mutants we identified as dominators really dominators? (regardless of whether we found them all)

How many of the actual dominators did we identify? (regardless of how many we mistakenly identified as dominators)
The cal() DMSG

• We used the Advanced Combinatorial Testing System (ACTS) to generate a test set
  – Pairwise permutations of months and year types generated 90 test cases
  – Test set killed all 145 non-equivalent mutants
  – Test results yield the dynamic mutant subsumption graph (DMSG)
The cal() DMSG

<table>
<thead>
<tr>
<th></th>
<th>TMGS</th>
<th>DMSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>38</td>
<td>30</td>
</tr>
<tr>
<td>Dominator Nodes</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>Mutants Killed</td>
<td>145</td>
<td>145</td>
</tr>
<tr>
<td>Mutation Score</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Mutants</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>Dominator Precision</td>
<td>1.00</td>
<td>0.40</td>
</tr>
<tr>
<td>Dominator Recall</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Subsumption Precision</td>
<td>1.00</td>
<td>0.77</td>
</tr>
<tr>
<td>Subsumption Recall</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Number of Tests</td>
<td>n/a</td>
<td>90</td>
</tr>
</tbody>
</table>
Dynamic Approximation

- Subject to errors due to incomplete test set
- May group mutants together where a distinguishing test is missing
- May add unsound edges where a contradicting test is missing
DMSG Summary

• The DMSG may be useful for research purposes

• *But what we are doing is completely backwards!*
  
  – It is *useless* to kill all the mutants in order to determine which mutants we should kill

• Can we perform static analysis of mutants instead?
Symbolic Execution

- We use Directed Incremental Symbolic Execution (DiSE) to analyze mutants
  - Implemented using Java PathFinder (JPF) and Symbolic PathFinder (SPF) from NASA Langley Research Center
  - Symbolic execution analyzes program execution in terms of its symbolic (rather than actual) inputs
- Generates reachability predicates necessary to reach the mutated point in the code
- Generates symbolic return values for program execution
Statically Analyzing Mutants

1. Generate mutants using muJava (2)
2. Process mutants using DiSE/JPF (4)
3. Analyze mutant predicates for equivalence (5) and generate tests that kill non-equivalent mutants (6)
4. Analyze subsumption relationships (7) and generate tests that distinguish mutants and create the static mutant subsumption graph (SMSG) (8)
### The cal() SMSG

<table>
<thead>
<tr>
<th></th>
<th>TMGS</th>
<th>DMSG</th>
<th>MSG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes</strong></td>
<td>38</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td><strong>Dominator Nodes</strong></td>
<td>10</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td><strong>Mutants Killed</strong></td>
<td>145</td>
<td>145</td>
<td>42</td>
</tr>
<tr>
<td><strong>Mutation Score</strong></td>
<td>1.00</td>
<td>1.00</td>
<td>0.29</td>
</tr>
<tr>
<td><strong>Dominator Mutants</strong></td>
<td>15</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td><strong>Dominator Precision</strong></td>
<td>1.00</td>
<td>0.40</td>
<td>0.60</td>
</tr>
<tr>
<td><strong>Dominator Recall</strong></td>
<td>1.00</td>
<td>1.00</td>
<td>0.20</td>
</tr>
<tr>
<td><strong>Subsumption Precision</strong></td>
<td>1.00</td>
<td>0.77</td>
<td>0.19</td>
</tr>
<tr>
<td><strong>Subsumption Recall</strong></td>
<td>1.00</td>
<td>1.00</td>
<td>0.39</td>
</tr>
<tr>
<td><strong>Number of Tests</strong></td>
<td>n/a</td>
<td>90</td>
<td>n/a</td>
</tr>
</tbody>
</table>
SMSG results for cal() 

- Failed to kill many mutants ✗
- Complexity is low ✗
- Refined a few relationships that the DMSG missed! ✓
  - AORB_3 ➡️ AORB_2
Static Approximation

- Dynamic analysis was never incorrect, only incomplete
- Static analysis is subject to errors due to *unsound analysis* or use of heuristics (loops, arrays, etc.)
- Distinguishes mutants that are very difficult to distinguish dynamically ✅
  - But – *may group mutants together where analysis can’t distinguish them* ❌
  - But – *may falsely distinguish indistinguishable mutants* ❌
- Finds subsumption relationships that are very difficult to determine dynamically ✅
  - But – *may fail to show legitimate subsumption relationships* ❌
  - But – *may show false subsumption relationships* ❌
- What can we do with this information?
Static Test Generation

• Each time the static analysis solver kills or distinguishes a mutant, it generates a test case as a proof
• Static analysis of cal() generated 335 unique dynamic tests
• We can perform our usual dynamic analysis using these tests
Statically-derived DMSG

<table>
<thead>
<tr>
<th></th>
<th>TMGS</th>
<th>DMSG</th>
<th>SMSG</th>
<th>SDMSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>38</td>
<td>30</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>Dominator Nodes</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Mutants Killed</td>
<td>145</td>
<td>145</td>
<td>42</td>
<td>142</td>
</tr>
<tr>
<td>Mutation Score</td>
<td>1.00</td>
<td>1.00</td>
<td>0.29</td>
<td>0.98</td>
</tr>
<tr>
<td>Dominator Mutants</td>
<td>15</td>
<td>25</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Dominator Precision</td>
<td>1.00</td>
<td>0.40</td>
<td>0.60</td>
<td>1.00</td>
</tr>
<tr>
<td>Dominator Recall</td>
<td>1.00</td>
<td>1.00</td>
<td>0.20</td>
<td>0.73</td>
</tr>
<tr>
<td>Subsumption Precision</td>
<td>1.00</td>
<td>0.77</td>
<td>0.19</td>
<td>0.71</td>
</tr>
<tr>
<td>Subsumption Recall</td>
<td>1.00</td>
<td>1.00</td>
<td>0.39</td>
<td>0.99</td>
</tr>
<tr>
<td>Number of Tests</td>
<td>n/a</td>
<td>90</td>
<td>n/a</td>
<td>335</td>
</tr>
</tbody>
</table>
Statically-derived DMSG

- Very close in complexity to the DMSG
- Contains refinements from static analysis
- **No manual effort!**
Summary

- Symbolic execution of mutants has significant limitations, but can directly produce a marginal approximation of the TMSG.
- By executing statically-derived tests dynamically, we can eliminate the errors caused by symbolic execution’s unsound analysis and get very close to the TMSG without any manual test generation.
Future Work

• How will this approach scale up to non-trivial software components?
• Can we improve our static analysis results through Dynamic Symbolic Execution?
Questions?

rkurtz2@gmu.edu
Backup Slides
Mapping Results to the TMSG

Static-derived tests kill all but 3 mutants
Precision and Recall

\[
\text{precision} = \frac{\text{truePositives}}{\text{truePositives} + \text{falsePositives}}
\]

\[
\text{recall} = \frac{\text{truePositives}}{\text{truePositives} + \text{falseNegatives}}
\]

- \textit{truePositives}: the number of dominator mutants properly identified as dominators
- \textit{falsePositives}: the number of subsumed mutants falsely identified as dominators
- \textit{falseNegatives}: the number of dominators falsely identified as subsumed mutants
Equivalence Algorithm

// Iterate for each pair of PCs from A and Mi.
for each PC from A
  for each PC from Mi
    // Can reachability constraints for A and Mi be
    // mutually satisfied for this pair of PCs?
    if satisfiable(RC_A && RC_Mi) then
      // Reachability satisfied. Is there is any solution
      // where the return values for A and Mi differ?
      if satisfiable(RC_A && RC_Mi && (RV_A != RV_Mi)) then
        // Reachability is satisfied and the return values
        // differ, Mi is killed. Save the solution.
        Mi is statically killed;
        save solution as test that kills Mi;
        return;
      end if
    end if
  end for
// Is any reachability constraint for Mi not matched with
// a constraint for A? If any (see footnote) exist then
// Mi cannot be evaluated.
if no satisfiable RC_Mi found for this RC_A then
  Mi cannot be evaluated;
end if
Subsumption Algorithm

// Iterate through each path condition from the original.
for each PC from A

// Does Mi subsume Mj? Iterate through each PC for Mi and determine whether the reachability constraint for the original and Mi can be mutually satisfied and the return values differ (Mi is killed).
for each PC from Mi
    if satisfiable(RC_A && RC_Mi && (RV_A != RV_Mi)) then
        // Mi is killed. Iterate through each PC for Mj and try to find a solution where the constraints for the original and both mutants are satisfied and where mutant Mi is NOT killed.
        for each PC from Mj
            if satisfiable(RC_A && RC_Mi && RC_Mj && (RV_A == RV_Mi)) then
                // Found a solution. Now try to find a solution where the constraints for all 3 are satisfied and Mi is killed and Mj is not killed. If found then Mi cannot subsume Mj.
                if satisfiable(RC_A && RC_Mi && RC_Mj && (RV_A != RV_Mi) && (RV_A == RV_Mj)) then
                    Mi cannot subsume Mj;
                end if
            end if
        end for
    end if
end for

// If no case where constraints for all 3 are satisfied, then Mi cannot subsume Mj.
if no satisfiable RC_Mj found for RC_A && RC_Mi then
    Mi cannot subsume Mj;
end if

// Repeat steps above to evaluate Mj -> Mi

// Based on findings, determine Mi -> Mj or Mj -> Mi.
if (Mi cannot subsume Mj) && (Mj cannot subsume Mi) then
    no subsumption relationship between Mi and Mj;
else if (Mi cannot subsume Mj) then
    Mj subsumes Mi;
else if (Mj cannot subsume Mi) then
    Mi subsumes Mj;
else // Mi subsumes Mj and Mj subsumes Mi.
    Mi and Mj are statically indistinguishable;
end if