Static Analysis of Mutant Subsumption

April 13, 2015

Bob Kurtz, *Paul Ammann, Jeff Offutt*

George Mason University

Contributions

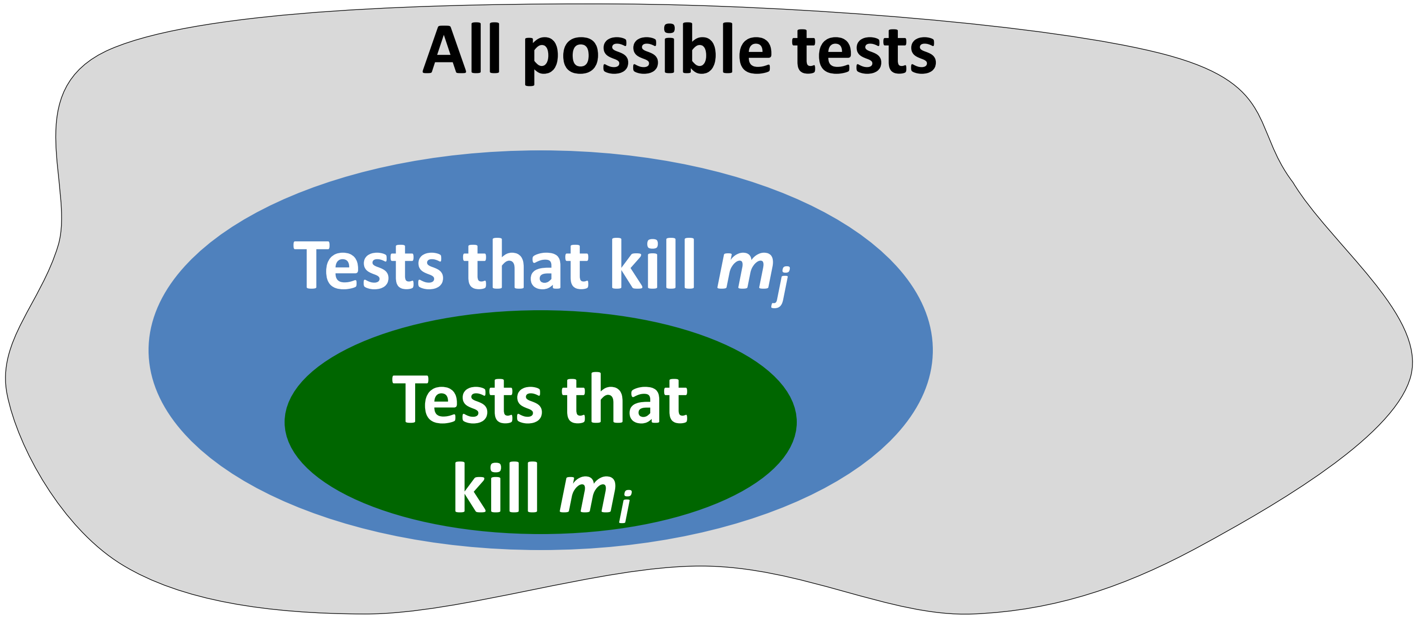
- We developed a process to determine mutant subsumption relationships statically using symbolic execution
- We used this process to generate test cases to kill mutants for a simple example program
- We fed these tests back into dynamic analysis to remove the “unsoundness” that is characteristic of symbolic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

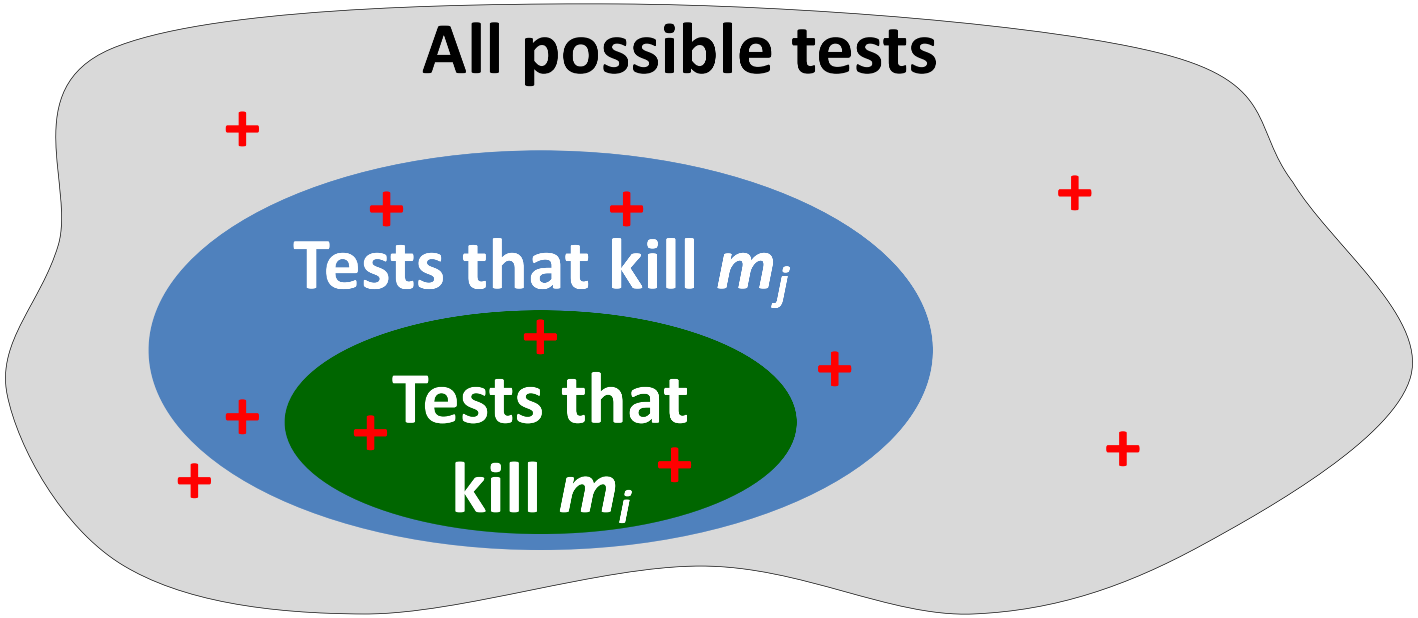
“True” Subsumption

- 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



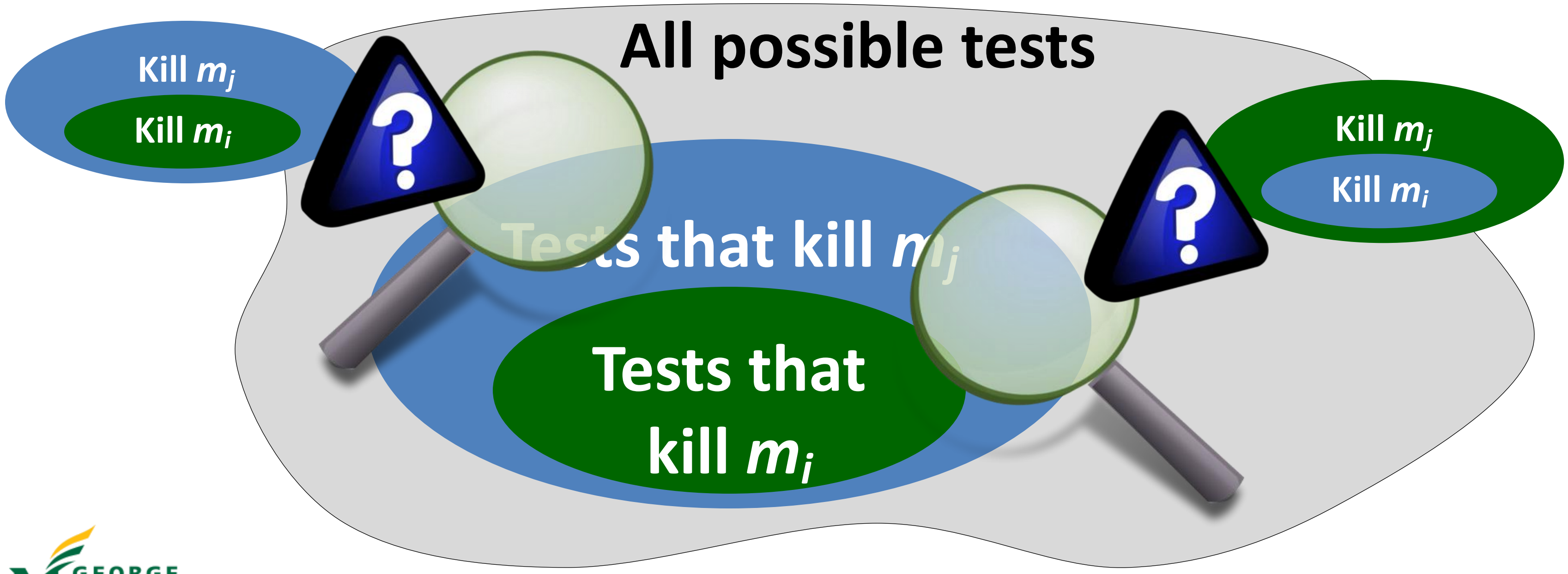
Dynamic Subsumption

- 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

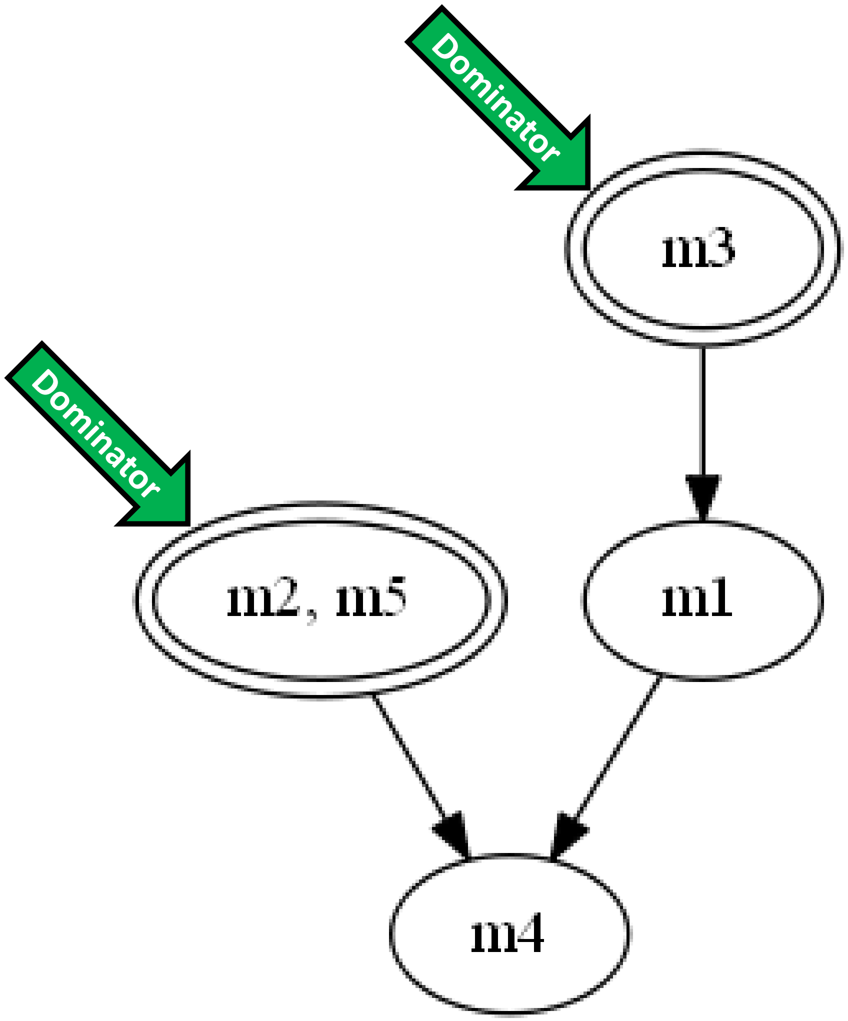


Static Subsumption

- 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!

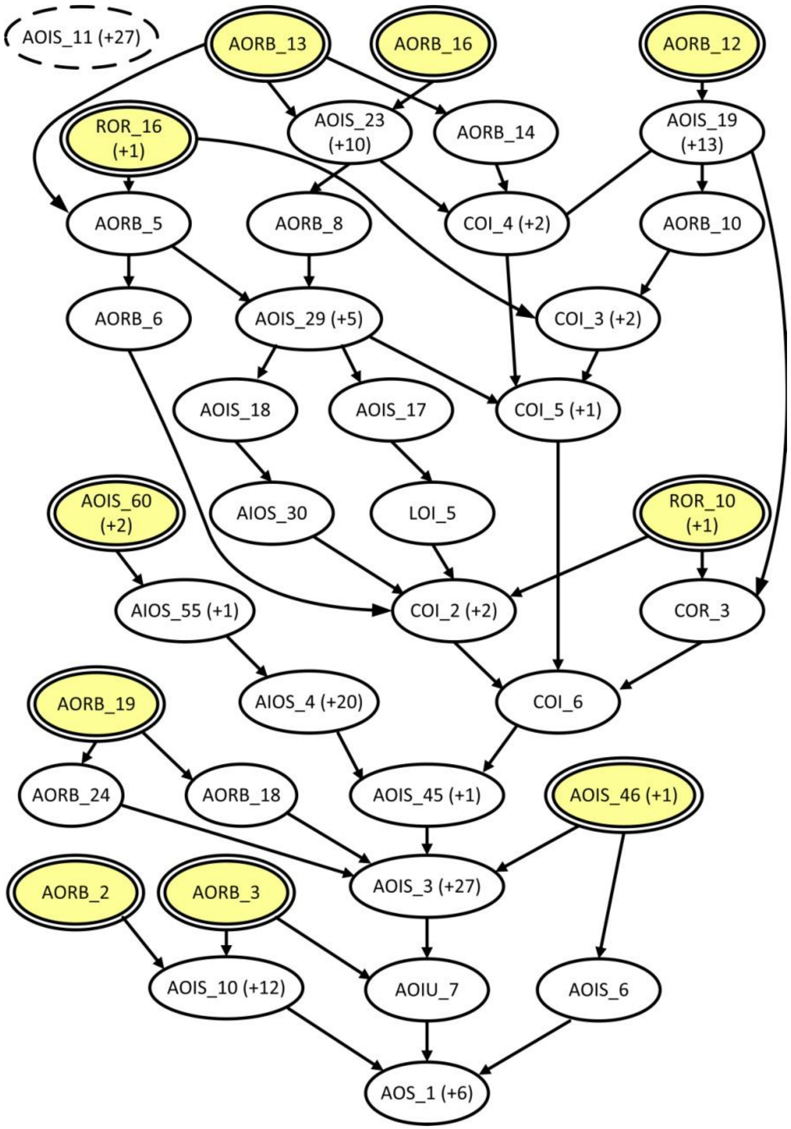
The cal() Example

- 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

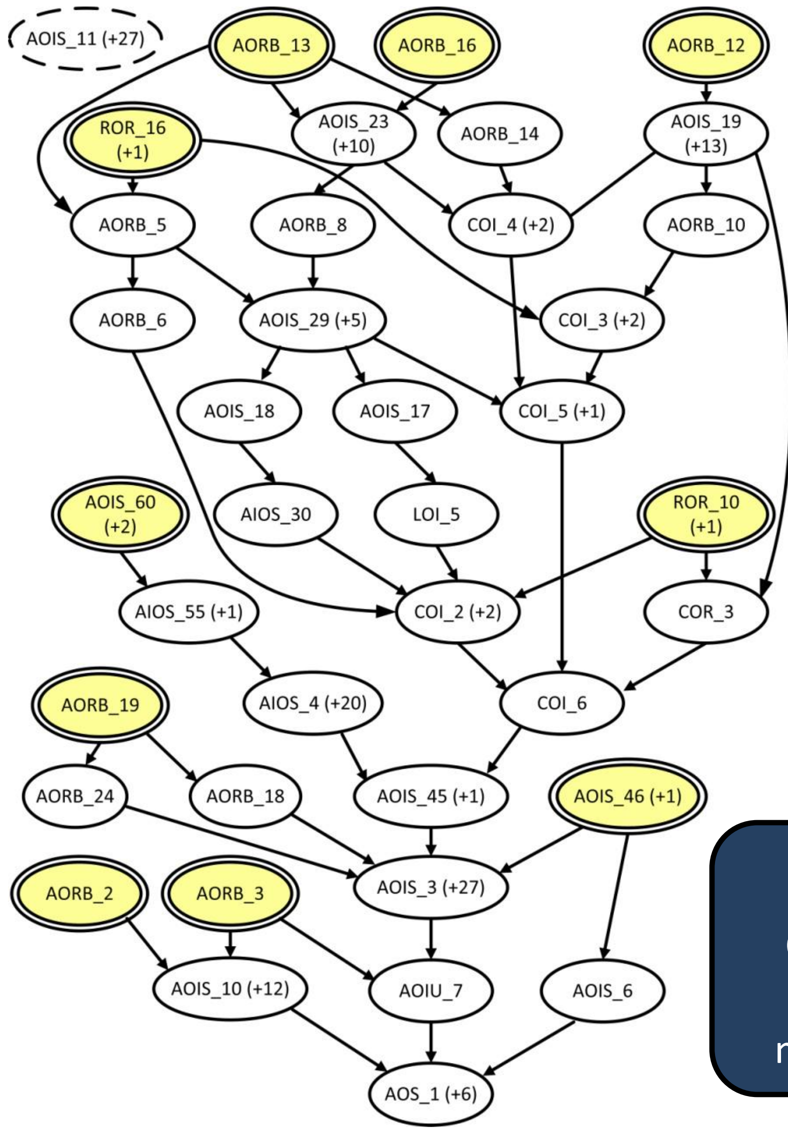
- We expected our various analyses to have limitations
- For comparison, we manually analyzed all 173 mutants and (with the help of a solver) determined their subsumption relationships
- The result is the ***true mutant subsumption graph*** (TMSG) for cal()

The cal() TMSG



	TMSG
Nodes	**38**
Dominator Nodes	**10**
Mutants Killed	**145**
Mutation Score	**1.00**
Dominator Mutants	**15**
Dominator Precision	**1.00**
Dominator Recall	**1.00**
Subsumption Precision	**1.00**
Subsumption Recall	**1.00**
Number of Tests	**n/a**

The cal() TMSG



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

Killed	**145**
Mu Score	**1.00**
Domin r Mutants	**15**
Dominator Precision	**1.00**
Dominator Recall	**1.00**
Subsumpti recision	**1.00**
Subsumpti ll	**1.00**
	n/a

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

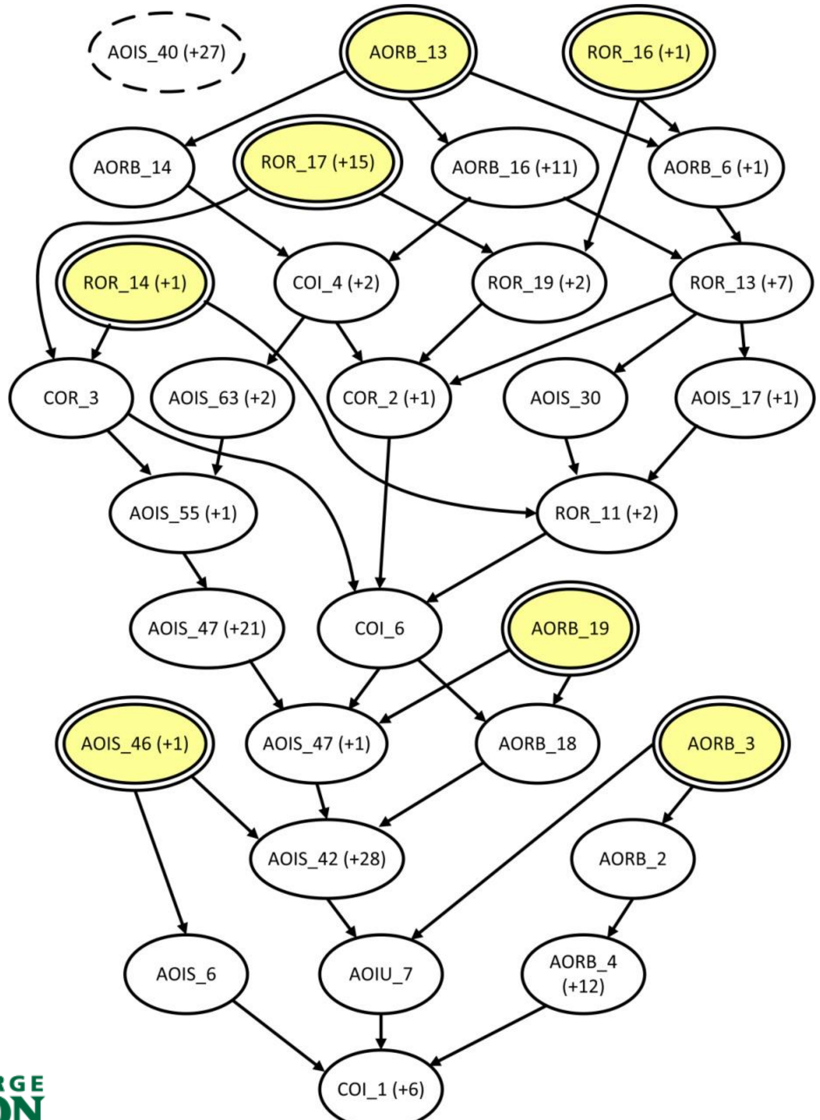
GEORGE MASON UNIVERSITY
WHERE INNOVATION IS TRADITION

Raytheon

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

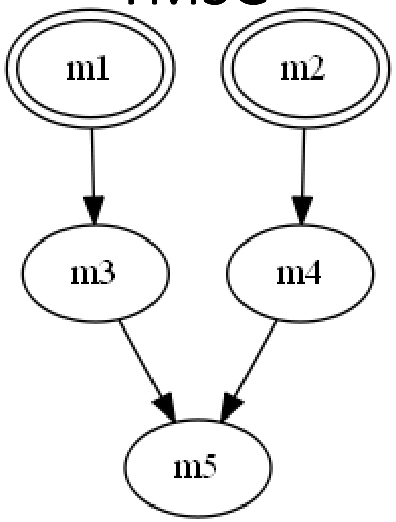


	TMGS	DMSG
Nodes	38	**30** ?
Dominator Nodes	10	**7** ?
Mutants Killed	145	**145** ✓
Mutation Score	1.00	**1.00** ✓
Dominator Mutants	15	**25** ✘
Dominator Precision	1.00	**0.40** ✘
Dominator Recall	1.00	**1.00** ✓
Subsumption Precision	1.00	**0.77** ?
Subsumption Recall	1.00	**1.00** ✓
Number of Tests	n/a	**90**

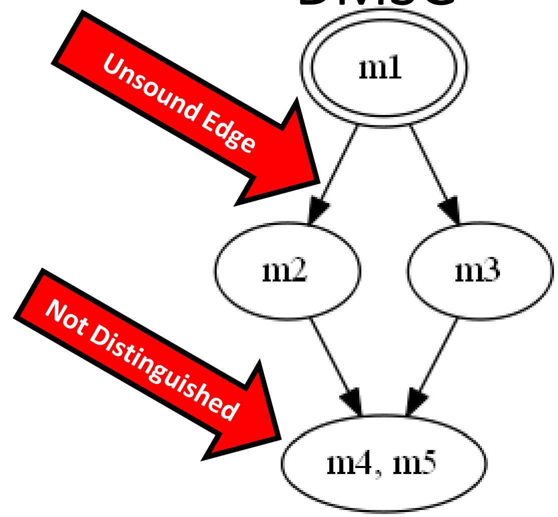
Dynamic Approximation

- 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

TMSG



DMSG



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

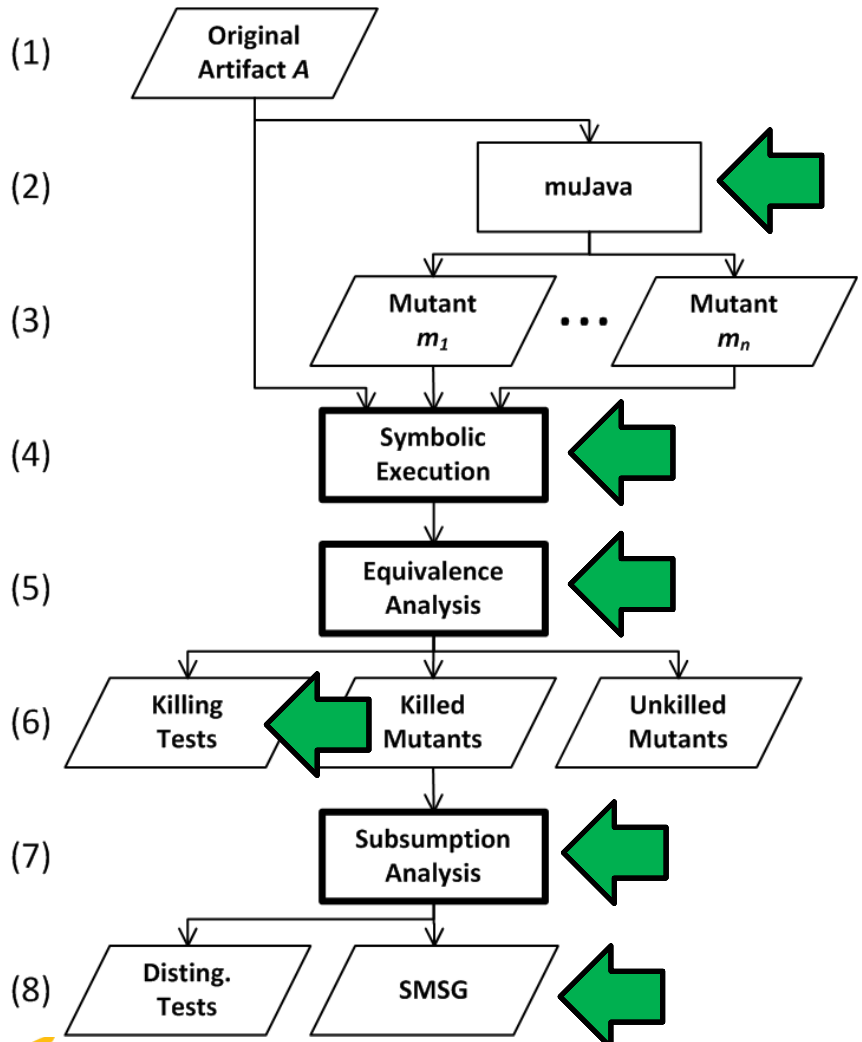
```
pc:( == RETURN ( / day2 day1))
&& ( != day1 0) && ( == month2 month1)

pc:( == RETURN ( + ( - 31 day1) day2))
&& ( > ( + month1 1) ( - month2 1))
&& ( != ( % year 400) 0)
&& ( == ( % year 100) 0)
&& ( == ( % year 4) 0)
&& ( != month2 month1)
```

Reachability predicates

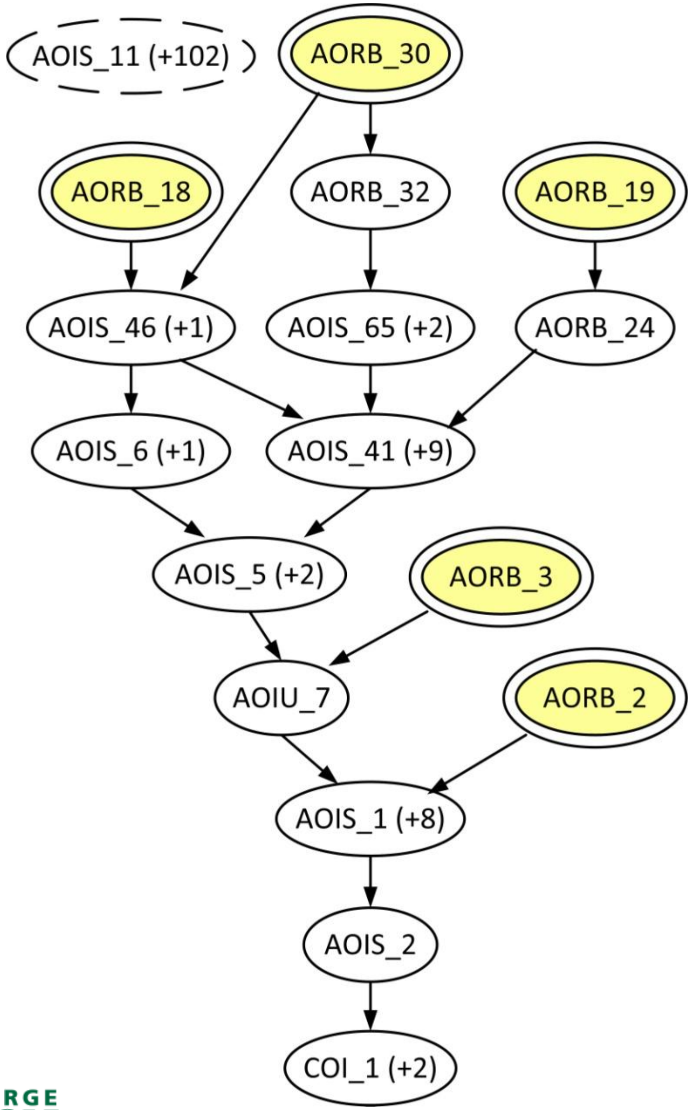
Return values

Statically Analyzing Mutants



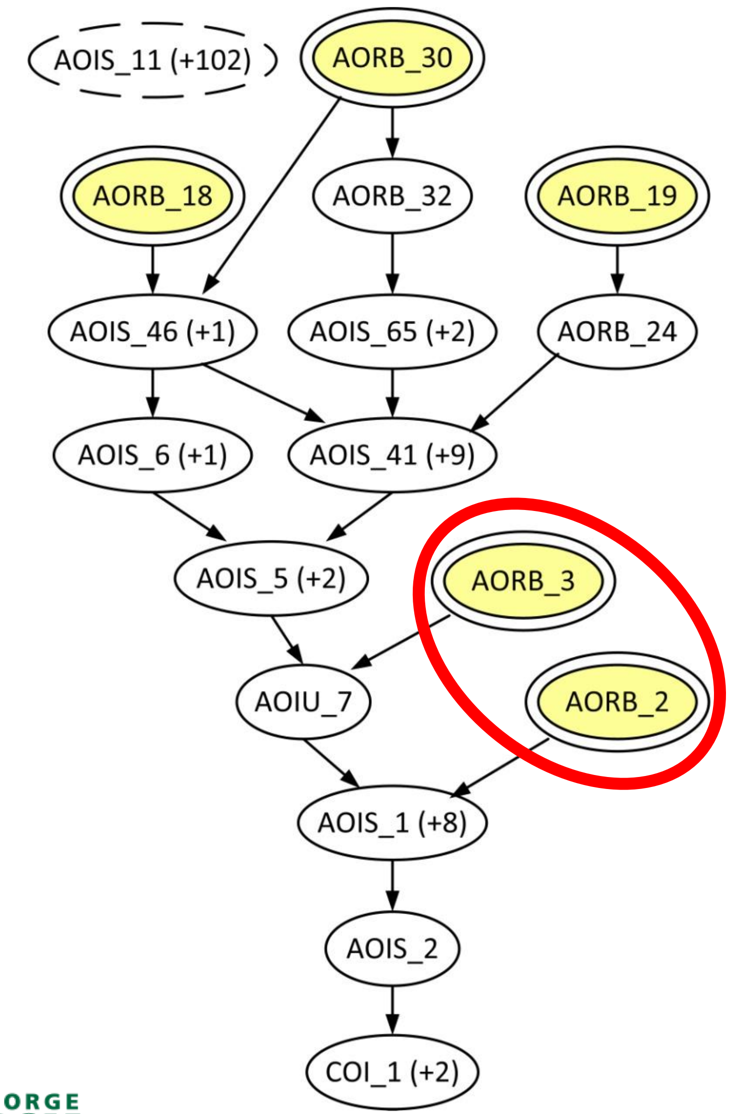
- Generate mutants using muJava (2)
- Process mutants using DiSE/JPF (4)
- Analyze mutant predicates for equivalance (5) and generate tests that kill non-equivalent mutants (6)
- Analyze subsumption relationships (7) and generate tests that distinguish mutants and create the ***static mutant subsumption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

The cal() SMSG



	TMGS	DMSG	SMSG
Nodes	38	30	**16**
Dominator Nodes	10	7	**5**
Mutants Killed	145	145	**42**
Mutation Score	1.00	1.00	**0.29**
Dominator Mutants	15	25	**5**
Dominator Precision	1.00	0.40	**0.60** ?
Dominator Recall	1.00	1.00	**0.20**
Subsumption Precision	1.00	0.77	**0.19**
Subsumption Recall	1.00	1.00	**0.39**
Number of Tests	n/a	90	**n/a**

SMSG results for cal()



- Failed to kill many mutants ✘
- Complexity is low ✘

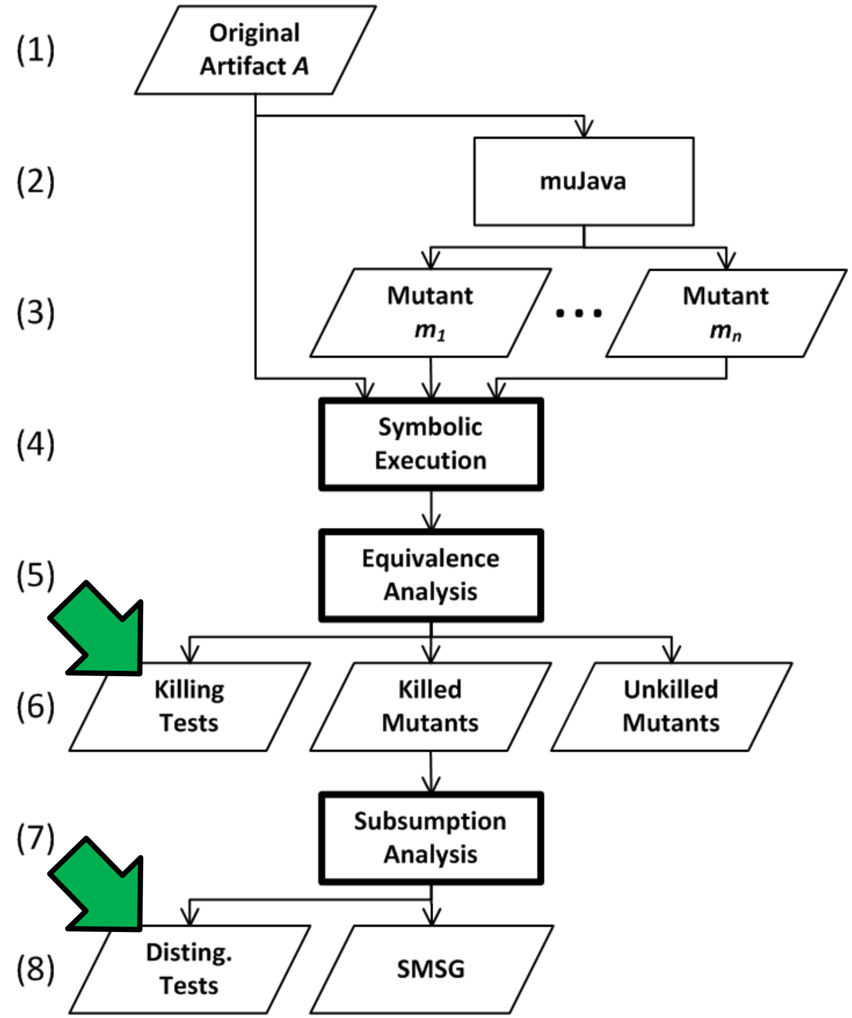
- Refined a few relationships that the DMSG missed! ✓
  - **AORB_3 $\not\rightarrow$ AORB_2**

Static Approximation

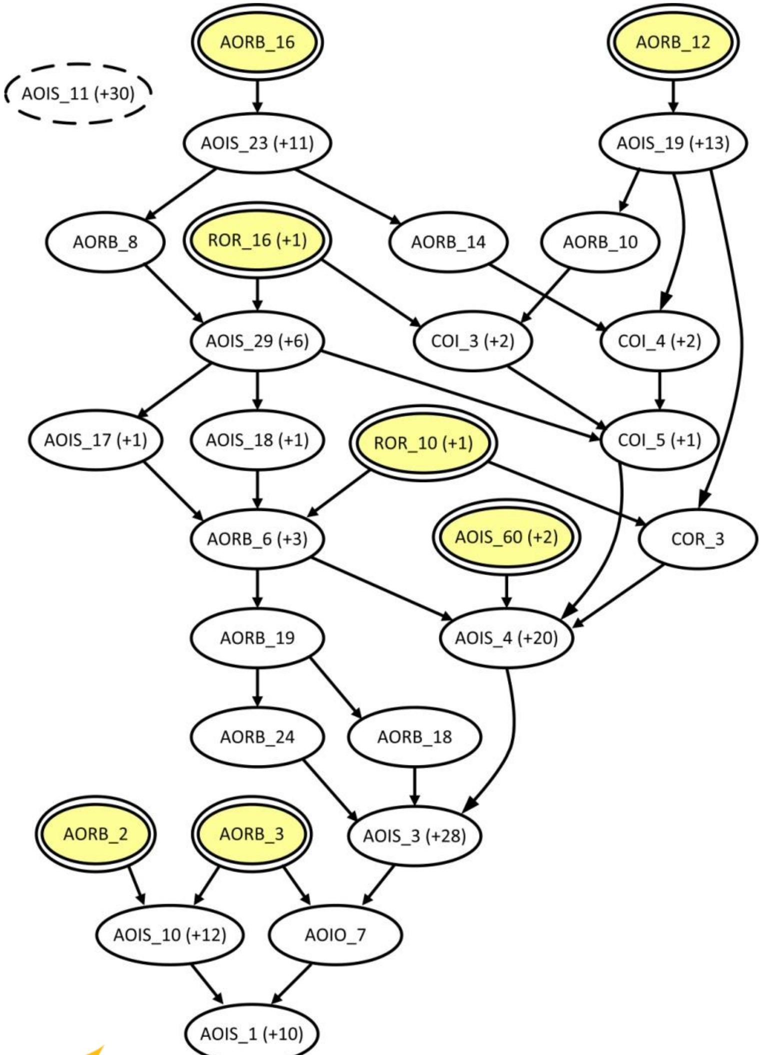
- Dynamic analysis was never incorrect, only incomplete
- Static analysis is subject to errors due to ***unsound analysis*** or use of heuristics (loops, arrays, etc.)
- Distinguishes mutants that are very difficult to distinguish dynamically ✓
  - *But – may group mutants together where analysis can't 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

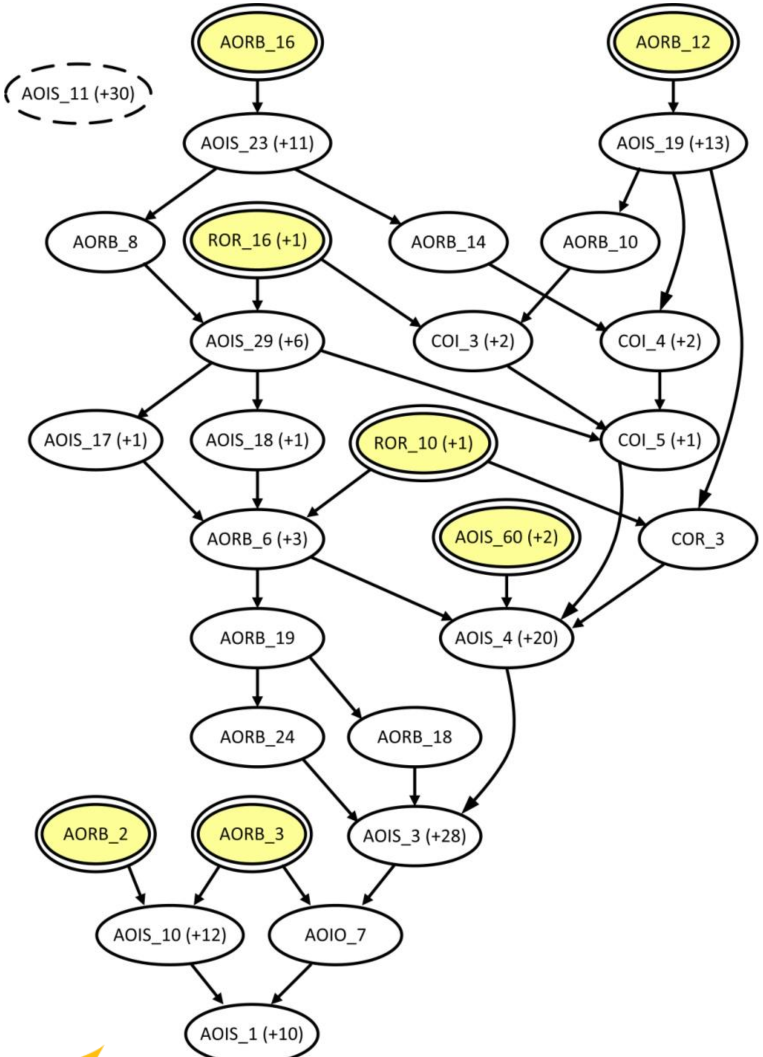


Statically-derived DMSG



	TMGS	DMSG	SMSG	SDMSG
Nodes	38	30	16	**28** ?
Dominator Nodes	10	7	5	**7** ?
Mutants Killed	145	145	42	**142** ✓
Mutation Score	1.00	1.00	0.29	**0.98** ✓
Dominator Mutants	15	25	5	**11** ✓
Dominator Precision	1.00	0.40	0.60	**1.00** ✓
Dominator Recall	1.00	1.00	0.20	**0.73** ?
Subsumption Precision	1.00	0.77	0.19	**0.71** ?
Subsumption Recall	1.00	1.00	0.39	**0.99** ✓
Number of Tests	n/a	90	n/a	**335**

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?

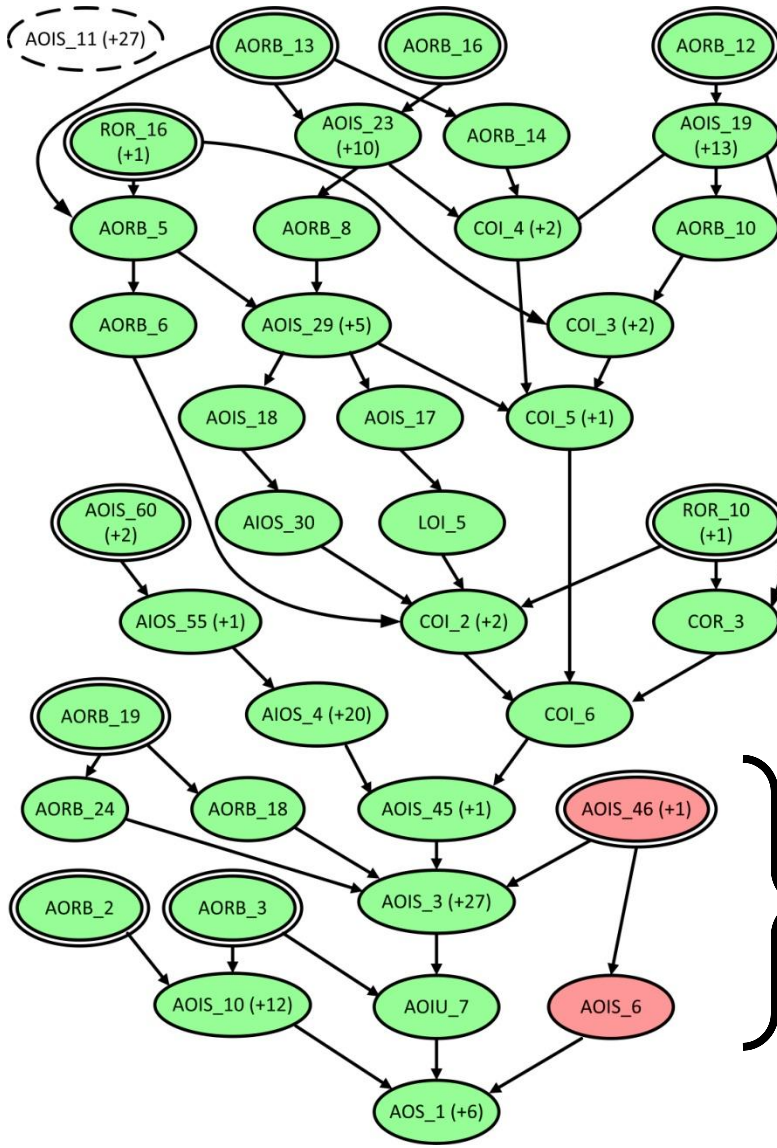
rkurtz2@gmu.edu

Backup Slides

GEORGE MASON UNIVERSITY
WHERE INNOVATION IS TRADITION

Raytheon

Mapping Results to the TMSG



Static-derived tests kill all but 3 mutants

Precision and Recall

$$precision = \frac{truePositives}{truePositives + falsePositives}$$

$$recall = \frac{truePositives}{truePositives + falseNegatives}$$

- *truePositives*: the number of dominator mutants properly identified as dominators
- *falsePositives*: the number of subsumed mutants falsely identified as dominators
- *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
end for
```

Subsumption Algorithm

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

  // Does Mi subsumes 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

      // 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
    end if
  end for

  // Repeat steps above to evaluate Mj -> Mi
    ...
end for

// 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
```