lab-soft
software engineering laboratory

An Analysis about Mutant Subsumption in First- and Second-Order Mutation Testing

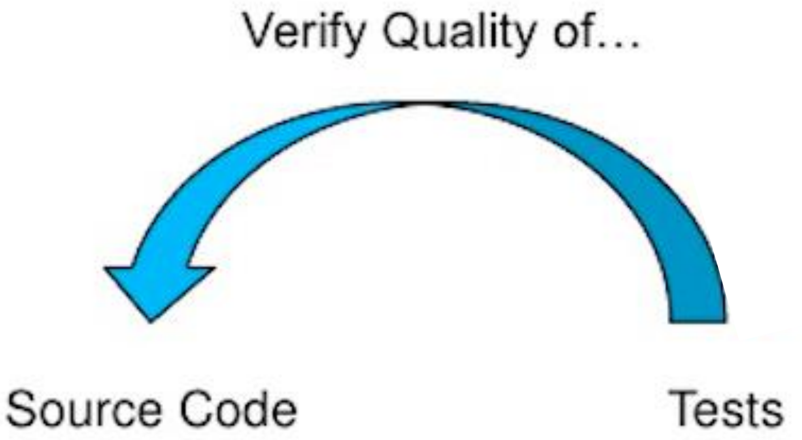
João Paulo de Freitas Diniz

LabSoft Seminar. Nov 22nd, 2024

Outline

- Mutation testing
- Mutants subsumption
- Dynamic mutant subsumption graphs
- Second-order mutants subsumption
- Study design
- Preliminary results
- Comparison with SS2OMs reduction
- Conclusion

Introduction



Mutation Testing

- Introducing **artificial syntactic changes** (**mutations**) into original source code
  - Intending to represent real common programming bugs
  - Changed programs are called **mutants**
- Running test cases on mutants
  - Result different from original: mutant **killed**
  - Otherwise: **alive**

Example of a mutant

Mutation place:

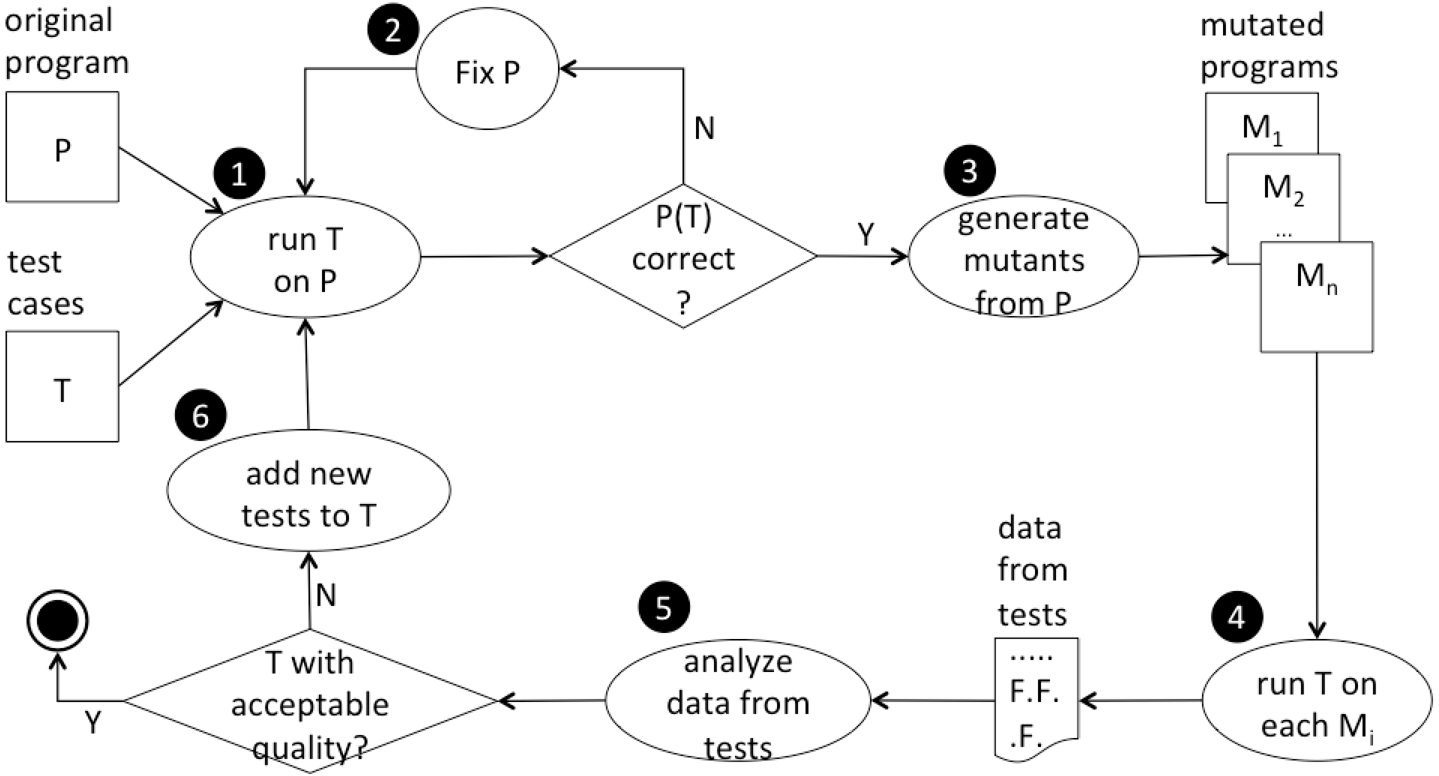
```
public class Taxes {

    double simpleTax(double amount) {

        return amount * 0.2;
    }
}
```

Example of a mutant

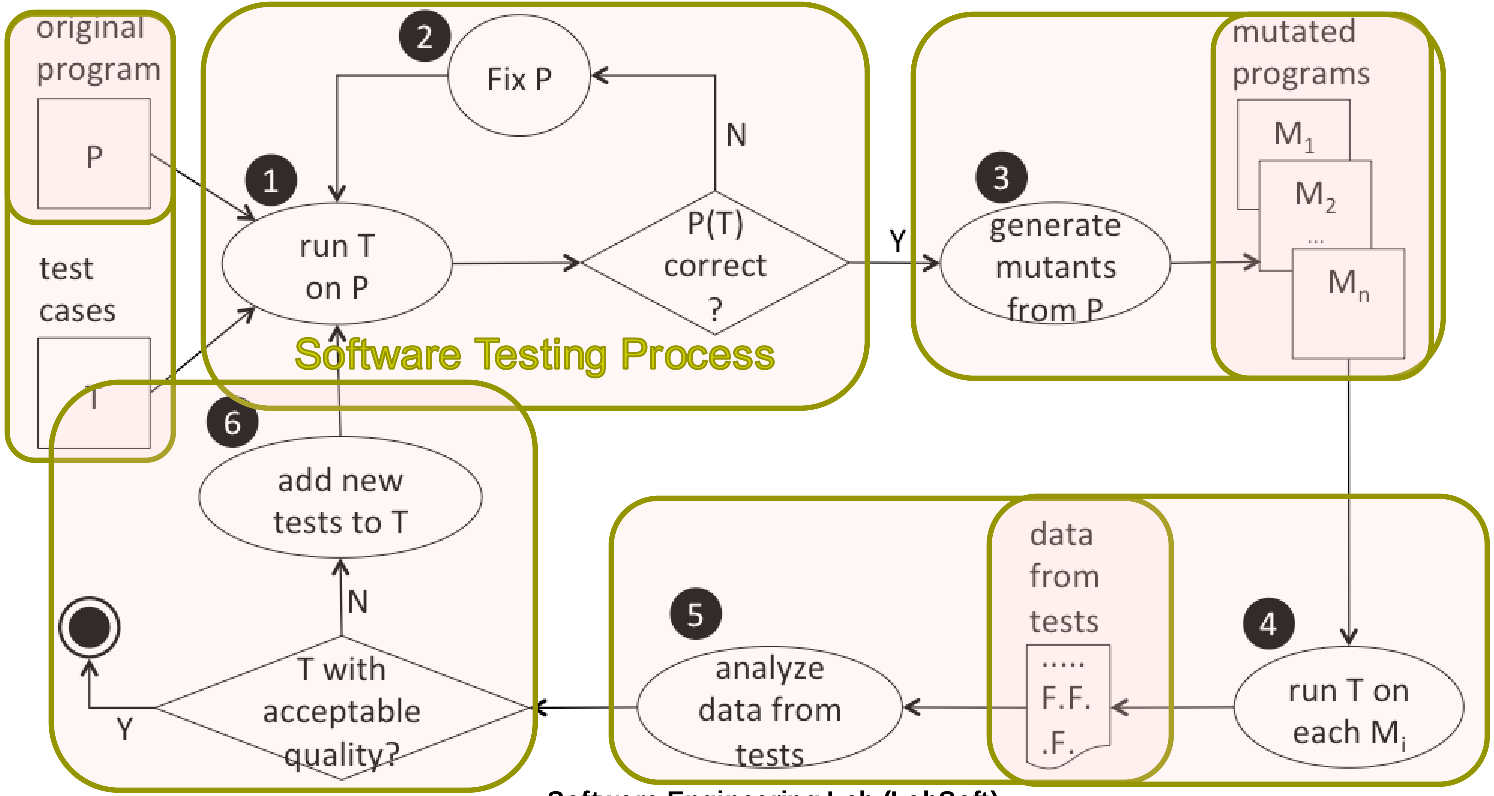
$* \longrightarrow +$

```
public class Taxes {
    double simpleTax(double amount) {
        return amount + 0.2;
    }
}
```

Mutation testing process

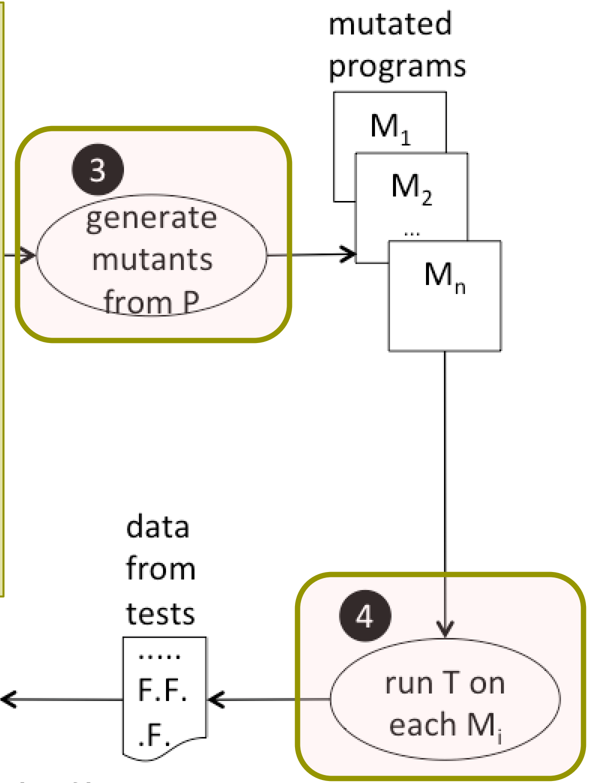


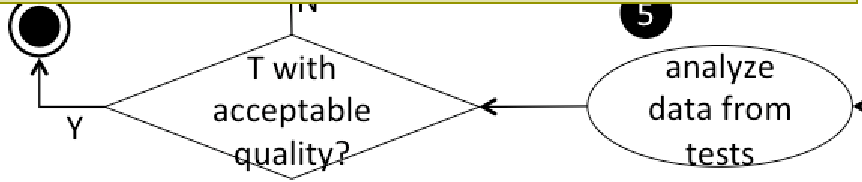
Mutation testing process



Mutation testing drawbacks

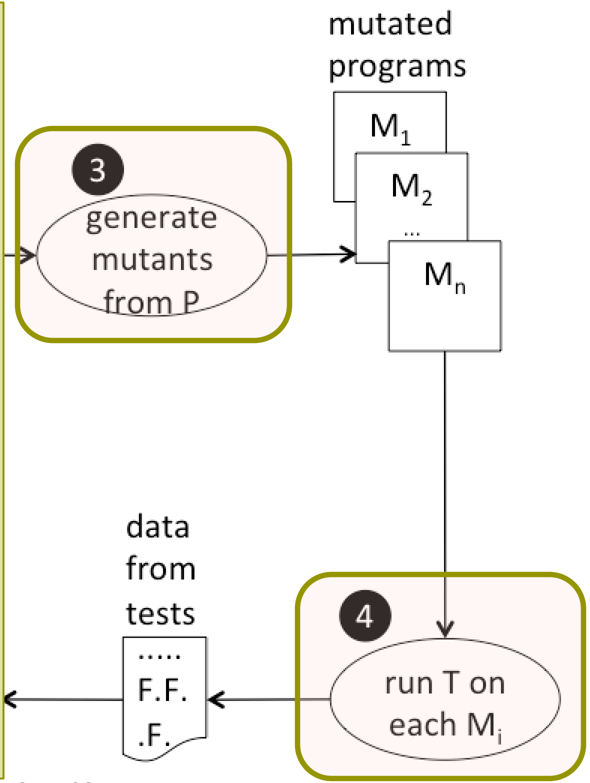
3. **Many mutable places**
   - Mutants generation
   - Compilation
4. **High computational cost**
   - Running tests





Mutation testing drawbacks

- Cost reduction techniques
  - Number of test cases
  - Test case prioritization
  - Number of mutants
    - **subsumption**



Mutants subsumption

Contextualization

```
def greaterThan(a, b):
    return a > b   # original
    return a >= b  # mutant 1
    return a <= b  # mutant 2
```

Contextualization

```
def greaterThan(a, b):
    return a > b   # original
    return a >= b  # mutant 1
    return a <= b  # mutant 2
```

	Test	orig
t1	assertTrue(greaterThan(6, 5))	✅
t2	assertFalse(greaterThan(5, 5))	✅
t3	assertFalse(greaterThan(5, 6))	✅

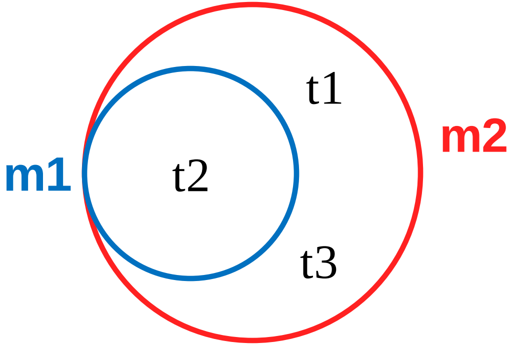
Contextualization

```
def greaterThan(a, b):
    return a > b    # original
    return a >= b # mutant 1
    return a <= b # mutant 2
```

	Test	orig	m1	m2
t1	`assertTrue(greaterThan(6, 5))`	✅	✅	❌
t2	`assertFalse(greaterThan(5, 5))`	✅	❌	❌
t3	`assertFalse(greaterThan(5, 6))`	✅	✅	❌

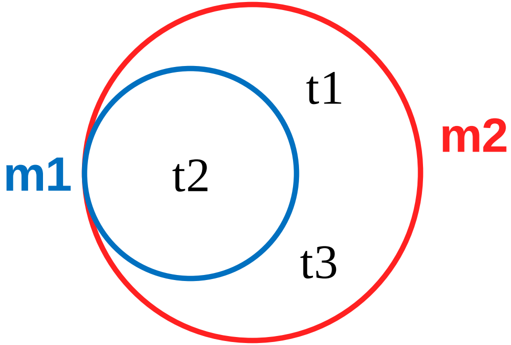
Contextualization

- Killing tests



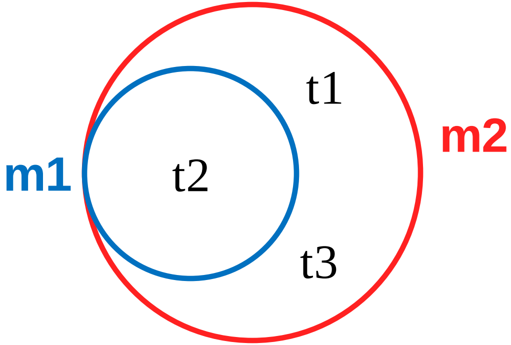
Contextualization

- All test sets that kill **m1** also kill **m2**



Conclusion

- **m1 subsumes m2**



Conclusion

- If we know beforehand that
  - **m1** subsumes **m2**
- Therefore,
  - **m2** should not have been generated

Cost reduction: fewer mutants to run the test suite against

Dynamic mutant subsumption graphs

Example

test	m1	m2	m3	m4	m5
t1	✗	✗		✗	✗
t2	✗		✗	✗	
t3				✗	
t4		✗		✗	✗

Subsumption relationships

test	m1	m2	m3	m4	m5
t1	✗	✗		✗	✗
t2	✗		✗	✗	
t3				✗	
t4		✗		✗	✗

m1 → m4

m2 → m4

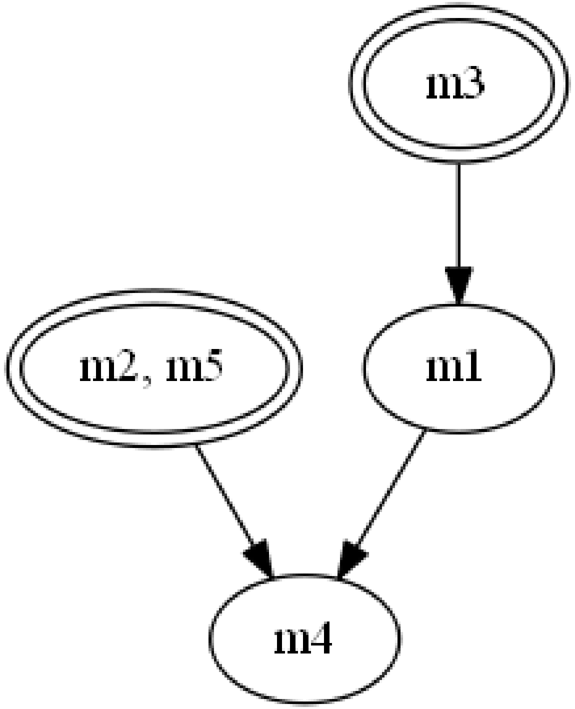
m3 → m1

m3 → m4

m5 → m4

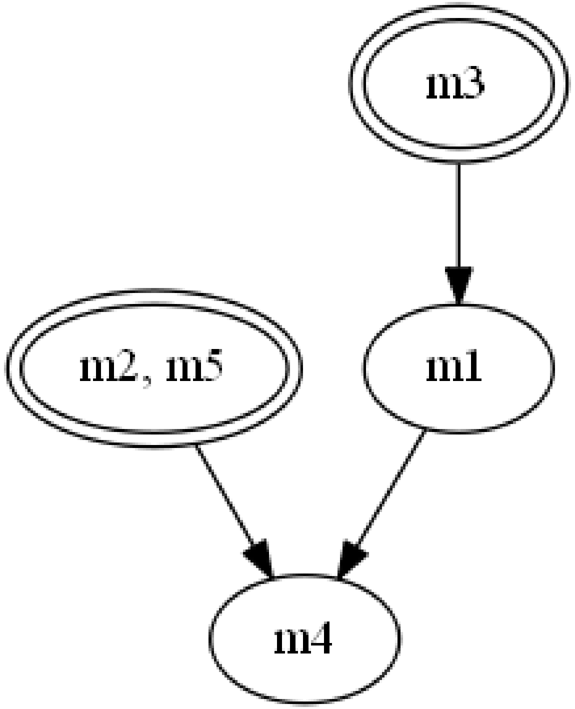
Subsumption graph

Test	m1	m2	m3	m4	m5
t1	✗	✗		✗	✗
t2	✗		✗	✗	
t3				✗	
t4		✗		✗	✗



Conclusion

- **Root nodes are kept**
  - 2 minimal
  - 3 mutants
- Remaining nodes
  - are disregarded
  - (redundants)



Second-order mutants subsumption

A 2OM subsumption example

Original

```
bool f(int a, int b):
  if (a == 1):
    return a < b
  return a > b
```

Mutation 1 (FOM)

```
bool f(int a, int b):
  if (a != 1):
    return a < b
  return a > b
```

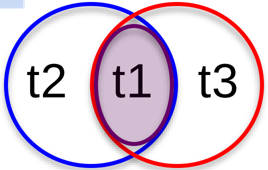
Mutation 2 (FOM)

```
bool f(int a, int b):
  if (a == 1):
    return a >= b
  return a > b
```

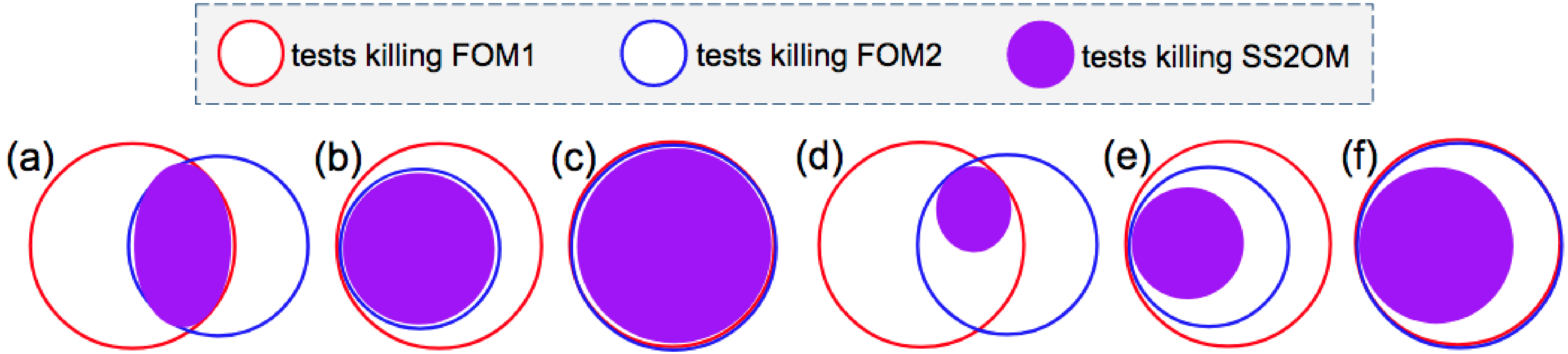
Both (HOM)

```
bool f(int a, int b):
  if (a != 1):
    return a >= b
  return a > b
```

Test	Original	Mutation 1	Mutation 2	Both
`assert  f(1, 2)` ←t1	Pass	Fail	Fail	Fail
`assert !f(0, 3)` ←t2	Pass	Fail	Pass	Pass
`assert !f(1, 1)` ←t3	Pass	Pass	Fail	Pass



Venn Diagram “shapes”



Mutants reduction - example

FOMs	**SS2OMs**
m1	[m1,m2]
m2	[m3,m4]
m3	[m5,m6]
m4	
m5	
m6	
m7	
m8	

Mutants reduction - example

FOMs	**SS2OMs**	**Resulting mutants**
~~m1~~	[m1,m2]	[m1,m2]
~~m2~~	[m3,m4]	[m3,m4]
~~m3~~	[m5,m6]	[m5,m6]
~~m4~~		
~~m5~~		
~~m6~~		
m7		
m8		

Mutants reduction - example

FOMs	SS2OMs	Resulting mutants
~~m1~~	[m1,m2]	[m1,m2]
~~m2~~	[m3,m4]	[m3,m4]
~~m3~~	[m5,m6]	[m5,m6]
~~m4~~		m7
~~m5~~		m8
~~m6~~		
m7		
m8		

non-subsumed FOMs

Mutants reduction - example

FOMs	SS2OMs	Resulting mutants	Reduction
~~m1~~	[m1,m2]	[m1,m2]	3 out of 8 mutants
~~m2~~	[m3,m4]	[m3,m4]	(37.5%)
~~m3~~	[m5,m6]	[m5,m6]	
~~m4~~		m7	
~~m5~~		m8	
~~m6~~			
m7			
m8			

non-subsumed FOMs

Study design

Dataset: 9 Java systems

System	Version	LOC	# Tests	JUnit	#FOMs
Vending Machine	Exceptions	~100	35	4	**57**
Triangle	n/a	34	12	4	**138**
Monopoly	n/a	1,181	124	3	**866**
Commons CSV	1.8	~2k	325	4	**925**
Commons CLI	1.4	2,699	318	4	**1,082**
ECal	2003.10	3,626	224	3	**1,207**
Commons Validator	1.6	7,409	536	4	**3,197**
Gson	2.9.0	> 10k	1,089	3 and 4	**3,712**
Chess	n/a	4,924	930	3 and 4	**5,287**

~16.5K

Study steps

- Compute the killing tests for each mutant
- Generate the subsumption graph
- Retrieve the root (minimal) nodes
- Compare with SS2OMs subsumption

Preliminary results

Subsumption analysis

Overall	#FOMs	#minimal nodes	#remaining FOMs
9 systems	16,471	1,115	3,376

Highlight on Triangle

{91}
{65, 67, 68, 70, 75, 77, 80, 52, 85, 56, 57, 58, 59, 63}
{35, 37, 38, 39, 108, 112, 114, 115, 116, 118, 119}
{129, 130, 131, 46, 122, 123, 124, 127}
{11}
{62}
{79}
{5}
{96, 132, 136, 121, 106, 111}
{69}
{97, 99, 100, 101, 103, 104, 23, 24, 25, 26, 27, 28, 93}
{0}

#FOMs	#minimal nodes	#remaining FOMs
138	12	59

Comparison with SS2OMs reduction

1^{st}- and 2^{nd}-order reductions

Overall	FOMs	Via subsumption graph	Via SS2OMs
9 systems	16,471	77.35%	22.37%

RQ

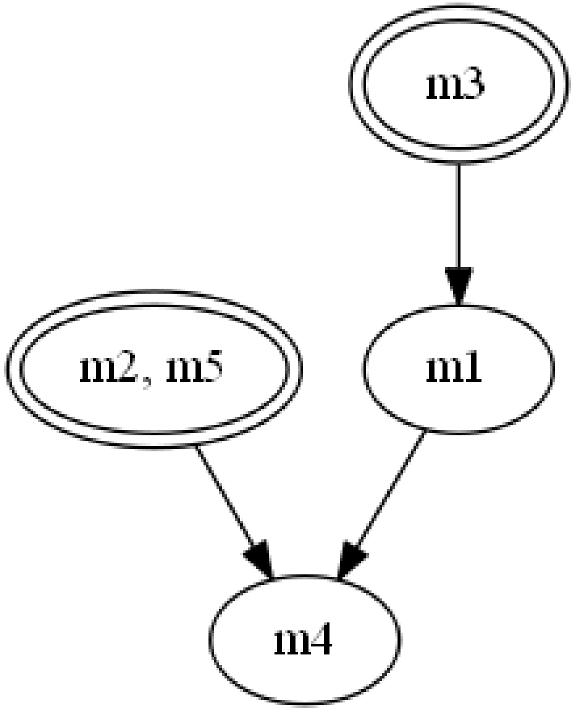
- Considering the **FOM** subsumption graphs, how are SS2OMs formed?

- Only by FOMs inside the **same** minimal nodes

  (not by FOMs of **distinct** minimal nodes)

RQ

- Consideri ɔsumption graphs, how are S l?



- Only by FC **ne** minimal nodes
  (not by minimal nodes)

Conclusion

- Can SS2OMs reduce even more mutants than the non-subsumed FOMs?

- **No, they do not contribute to any further reduction.**

Software Engineering Lab (LabSoft)
http://labsoft.dcc.ufmg.br/

Questions?