Symbolic methods for efficient mutation testing
(work in progress)

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle
Paris area, France

joint work with François Cheynier
Dynamic Symbolic Execution \([\text{sage, pex, klee, pathcrawler, osmose, \ldots}]\)

- ✓ very powerful approach to (white box) test generation
- × but tools often target very basic coverage criteria (instructions, decisions)

Mutation coverage criterion

- ✓ very powerful coverage criterion
- × but lacks automation
Two main goals

Use symbolic methods to automate mutation testing

- test data generation
- equivalent mutant detection
- efficient mutant-coverage computation [not so easy with mutants]

Leverage DSE methods to more powerful coverage criteria

Approach: try to reuse symbolic tools in a blackbox manner
Current results

- identify which part of (weak) mutations can be efficiently automated

- leverage DSE to mutation testing (blackbox)
  - need careful instrumentation and incremental use of DSE
  - achieve high-level mutation coverage with reasonable overhead
    [on a few examples]

- explore other sources of automation:
  - very efficient mutant-coverage computation (low overhead)
  - effective detection of equivalent mutants
Introduction

Mutation testing

From mutants to programs with labels

DSE and mutants

Other results

Conclusion
Mutation testing in a nutshell

Mutant $M = \text{syntactic modification of program } P$

- Statement $i-1$: $x := y+z$
- Statement $i+1$: $x := y+z$

- Statement $i-1$: $x := y*z$
- Statement $i+1$: $x := y+z$
Mutation testing in a nutshell

- Mutant $M$ = syntactic modification of program $P$
- Covering $M$ = finding a DT $t$ such that $P(t) \neq M(t)$
Difficulties for automation

Automatic mutant coverage

- have to reason on two different traces on two (slightly) different programs
- quite far from usual invariance / reachability problems

Efficient mutant-coverage computation (wrt. test set $T$)

- $|\mathcal{M}|$ compilations and $|T| \times (|\mathcal{M}| + 1)$ executions
  
  // standard $D$-coverage : 1 compilation, $|T|$ executions

Problem of equivalent mutants

- cannot be distinguished from P
- waste resources trying to cover them
Weak mutations

Program P

```
statement i-1;
x := y+z;
statement i+1;
```

Output P(t)

Mutant M

```
statement i-1;
x := y*z;
statement i+1;
```

Output M(t)

(test) mutation
Weak mutations

```
Program P

statement i-1;
x := y + z;
statement i+1;
```

```
Mutant M

statement i-1;
x := y * z;
statement i+1;
```

Test t

Output P(t)

Output M(t)

weak mutation

strong mutation
Weak mutations

- empirically: almost as powerful as strong mutations
- weak mutant coverage looks like predicate reachability
Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion
Push forward the (weak mutant ↔ predicate reachability) analogy

Hope = reuse verification tools for mutation testing
From weak mutants to labels

Program P

Mutant M1

statement i-1;
\[x := f(d);\]
y := e;
statement i+2;

Mutant M2

statement i-1;
\[x := d;\]
y := e;
statement i+2;

Program with label

statement i-1:
\[x := d;\]
y := e;
\[\//d \neq f(d)\]
\[\//e \neq g(e)\]
statement i+2;

labels:
- predicates
- no side-effects
- just monitoring
A few definitions

- label $l = \text{pair (} loc, pred \text{)}$
- program with label $= < P, L >$, with $L$ a set of labels
- a test data $t$ covers $l$: $P(t)$ reaches $loc$ and satisfies $pred$
- $TS \models_c P$ denotes "$TS$ fully covers criterion $C$ on $P$"

Theorem

For all program $P$ and side-effect free mutation operator $op$, we can deduce an annotated program $< P, L >$ such that:
for any test set $TS$, $TS \models L P$ iff $TS \models M(op) P$
A systematic transformation

**Mutation inside a statement**

- \( \text{lhs := e} \mapsto \text{lhs := e'} \)
- add label: \( e \neq e' \)

**Mutation inside a decision**

- \( \text{if (cond)} \ldots \mapsto \text{if (cond')} \ldots \)
- add label: \( \text{cond} \oplus \text{cond'} \)
A systematic transformation (2)

We can use simpler transformations in many cases

- $x \mapsto -x$: add label $x \neq 0$
- $x+y \mapsto x-y$: add label $y \neq 0$
- $x \geq y \mapsto x > y$: add label $x = y$
- ...

Labels and weak mutations

Strong analogies

Weak mutant $\leftrightarrow$ Label
Weakly killed (covered) mutant $\leftrightarrow$ Covered label
Mutation score $\leftrightarrow$ Label coverage rate
Equivalent mutant $\leftrightarrow$ Uncoverable labels

But label coverage is easier to automatize

- $\approx$ predicate reachability
- instrumentation $+$ reuse of verification tools
Toward automatic mutation testing

Program P

Mutation op.

Instrumentation step

Annotated Program $<P+L>$

Program $P'$

Test generation (DSE)

Equivalent checking (static analysis)

coverage measurement
Toward automatic mutation testing

Program P

Instrumentation step

Annotated Program \(<P+L>\)

Program P'

Mutation op.

Pathcrawler

Test generation (DSE)

Frama-C

Equivalent checking (static analysis)

coverage measurement
Direct instrumentation

covering label p \equiv covering branch True
Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion
Direct instrumentation is not good enough

- Launch the PathCrawler tool (PC) on a few instrumented P’
- Compare with speed of PC launch on P

<table>
<thead>
<tr>
<th></th>
<th>PC on P</th>
<th>PC on P’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(basic)</td>
<td>(instrumentation)</td>
</tr>
<tr>
<td>Trityp</td>
<td>50 LOC</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>141 labels</td>
<td>466 s</td>
</tr>
<tr>
<td></td>
<td>0 s</td>
<td>63 TC</td>
</tr>
<tr>
<td></td>
<td>14 TC</td>
<td></td>
</tr>
<tr>
<td>Replace</td>
<td>100 LOC</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>79 labels</td>
<td>1 745 s</td>
</tr>
<tr>
<td></td>
<td>2 s</td>
<td>275 TC</td>
</tr>
<tr>
<td></td>
<td>121 TC</td>
<td></td>
</tr>
<tr>
<td>TCas</td>
<td>124 LOC</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>111 labels</td>
<td>228 767 s</td>
</tr>
<tr>
<td></td>
<td>4 s</td>
<td>249 TC</td>
</tr>
<tr>
<td></td>
<td>164 TC</td>
<td></td>
</tr>
</tbody>
</table>
Direct instrumentation is not good enough

- Launch the PathCrawler tool (PC) on a few instrumented P’
- Compare with speed of PC launch on P

<table>
<thead>
<tr>
<th></th>
<th>PC on P (basic)</th>
<th>PC on P’ (instrumentation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trityp</td>
<td>50 LOC, 141 labels</td>
<td>0 s, 14 TC</td>
</tr>
<tr>
<td>Replace</td>
<td>100 LOC, 79 labels</td>
<td>2 s, 121 TC</td>
</tr>
<tr>
<td>TCas</td>
<td>124 LOC, 111 labels</td>
<td>4 s, 164 TC</td>
</tr>
</tbody>
</table>

✓ Achieve high mutation coverage
× Very significant overhead !!!
P’ has exponentially more paths than P

Paths in P’ are much more more complex than needed

- at each “label”, an execution $\sigma$ is required to cover $p$ or $\neg p$
- an execution $\sigma_1\sigma_2\ldots\sigma_n$ in P can become $\sigma_1(l_1)\sigma_2(\neg l_2)\ldots(l_n)\sigma_n$ in P’
Direct instrumentation is not good enough (2)

Direct instrumentation

1

2^N paths

p1

True False

2

pN

True False

N
Direct instrumentation is not good enough (2)
Optim 1: smarter instrumentation

✓ Paths in $P''$ have at most one "label constraint"
✓ $P''$ has exponentially less paths than $P'$
Optim 1: smarter instrumentation

Direct instrumentation

1
\rightarrow p1
\rightarrow 2
\rightarrow pN
\rightarrow \mathit{N}

True \quad \text{False}

Smarter Instrumentation

1
\rightarrow \text{non}_\text{det}()
\rightarrow \text{assert}(p1)
\rightarrow 2
\rightarrow \text{non}_\text{det}()
\rightarrow \text{assert}(pN)
\rightarrow \mathit{N}
Direct instrumentation

1

2

2^N paths

p1

True
False

N

Cycles

Smarter Instrumentation

1

N+1 paths

nondet()

assert(p1)

2

assert(pN)

N
Optim 1: smarter instrumentation

Direct instrumentation

1 → 2

\[ 2^N \text{ paths} \]

combination

p1

True → False

2 → pN

True → False

pN → N

Smarter Instrumentation

1 → non_det()

\[ N+1 \text{ paths} \]

no combination

assert(p1)

2 → non_det()

assert(pN)

no additional constraint

N
Optim 2: incremental DSE

Observations

- we need to cover each label only once
- yet, already covered labels “stay in place” in P’
  - burden the DSE tool with “useless” paths
Optim 2: incremental DSE

Observations

- we need to cover each label only once
- yet, already covered labels “stay in place” in $P'$
  - burden the DSE tool with “useless” paths

Our approach: incremental DSE

- Partition labels into disjoint sets $L_1, \ldots, L_k$
- Choose a set $L_k$
- Launch PathCrawler on $< P, L_k >$
- when terminates:
  - get the generated test set $TS$
  - prune from all $L_i$ the labels covered by $TS$
- loop forall $L_i$
### Results

<table>
<thead>
<tr>
<th>Function</th>
<th>LOC</th>
<th>Labels</th>
<th>Basic</th>
<th>Direct</th>
<th>Smart</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trityp</strong></td>
<td>50</td>
<td>141</td>
<td>0s</td>
<td>466s</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 TC</td>
<td>63 TC</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84 TC</td>
<td></td>
</tr>
<tr>
<td><strong>Replace</strong></td>
<td>100</td>
<td>79</td>
<td>2s</td>
<td>1745s</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>121 TC</td>
<td>275 TC</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>393 TC</td>
<td></td>
</tr>
<tr>
<td><strong>TCas</strong></td>
<td>124</td>
<td>111</td>
<td>4s</td>
<td>228 767s</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>164 TC</td>
<td>249 TC</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 049 TC</td>
<td></td>
</tr>
</tbody>
</table>

- ✓ higher mutation coverage
- ✓ significant speedup wrt. direct instrumentation
  - yet, significant overhead (20x) wrt. uninstrumented version
Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion
Coverage computation

- use Emma (decision coverage) and MuJava (mutation score)
- 100 tests

<table>
<thead>
<tr>
<th></th>
<th>Emma</th>
<th>MuJava</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P (witness)</td>
<td>P'</td>
</tr>
<tr>
<td>TCas</td>
<td>124 LOC 111 labels</td>
<td>0.03 s</td>
</tr>
<tr>
<td>Replace</td>
<td>436 LOC 607 labels</td>
<td>0.05 s</td>
</tr>
<tr>
<td>Jtopas</td>
<td>5400 LOC 610 labels</td>
<td>3.22 s</td>
</tr>
</tbody>
</table>

Results
- significant speed-up (x40 - x100)
- very reasonable overhead (x1 - x4)
Detection of uncoverable labels (≈ equiv. mutants)

We use the Frama-C framework to detect uncoverable labels

- Value Analysis plugin (VA)
- restricted logical reasoning (L) [ad hoc plugin]

<table>
<thead>
<tr>
<th></th>
<th># labels</th>
<th>Categories of uncovered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>uncovered</td>
</tr>
<tr>
<td>Trityp</td>
<td>141</td>
<td>6</td>
</tr>
<tr>
<td>TCas</td>
<td>111</td>
<td>13</td>
</tr>
<tr>
<td>Replace</td>
<td>79</td>
<td>12</td>
</tr>
</tbody>
</table>

VA (resp. L) : # labels proved uncoverable through VA (resp. L)
M : # labels manually proved uncoverable

Results [direct instrumentation]

- good detection rate (0.6 - 1) with generic-purpose analysis
- can probably be improved
Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion
Conclusion & future work

A substantial subset of weak mutations can be reduced to predicate reachability

Allow to reuse the whole (symbolic) “verification machinery”

We have shown first encouraging experiments for

- automatic test generation with high mutation score
- automatic discovery of equivalent mutants
- efficient coverage measurement

Future work

- more experiments
- native support for labels in PathCrawler (goal = low overhead)
- dedicated mutant equivalence checking plugin in Frama-C?