Advanced Test Coverage Criteria:
Specify and Measure, Cover and Unmask

Sébastien Bardin & Nikolai Kosmatov

joint work with
Omar Chebaro, Robin David, Mickaël Delahaye, Michaël Marcozzi,
Mike Papadakis, Virgile Prevosto, etc.

CEA LIST
Software Safety & Security Lab
(Paris-Saclay, France)
Dynamic Symbolic Execution (DSE) is great! [Klee also!]

✓ robust, no false alarm, scale

✗ But ...
Dynamic Symbolic Execution (DSE) is great! [Klee also!]

- robust, no false alarm, scale
- But ... no native support for coverage criteria
Dynamic Symbolic Execution (DSE) is great! [Klee also!]

✓ robust, no false alarm, scale

✗ But ... no native support for coverage criteria

DSE can be efficiently lifted to coverage-oriented testing

- unified view of coverage criteria [ICST 14, ICST 17]
- a dedicated variant DSE* [ICST 14]
- moreover: infeasibility detection is feasible [ICST 15, ICSE 18]

Prototype LTest (Frama-C plugin) [TAP 14]

- all-in-one toolkit for testing C programs
- combination of Frama-C and PathCrawler
White-box software testing

Testing process

- Generate a test input
- Run it and check for errors
- Estimate coverage: if enough stop, else loop

```
int myfun(int ...) {
  if (x <= b) {
    ....
  } else ...
  return ...;
}
```
White-box software testing

Testing process
- Generate a test input
- Run it and check for errors
- Estimate coverage:
  if enough stop, else loop

Coverage criteria [decision, mcdc, etc.]
- systematic way of deriving test objectives
- major role: guide testing, decide when to stop, assess quality
- can be part of industrial normative requirements
- beware: lots of different coverage criteria
- beware: infeasible test requirements
Coverage criteria in white-box testing

Variety and sophistication gap between literature and testing tools

**Literature:**
- 28 various white-box criteria in the Ammann & Offutt book

**Tools:**
- restricted to small subsets of criteria
- extension is complex and costly
Another enemy: uncoverable test objectives

- waste generation effort, imprecise coverage ratios
- reason: structural coverage criteria are ... structural
- detecting uncoverable test objectives is undecidable

Recognized as a hard and important issue in testing

- no practical solution
- not so much work (compared to test gen.)
- real pain (e.g. aeronautics, mutation testing)
Extend DSE to advanced coverage criteria

- in an efficient way
- in a unified way
Goals and Challenges

Extend DSE to advanced coverage criteria

- in an efficient way
- in a unified way

Not easy! [Active Testing, Augmented DSE, Mutation DSE]

- limited or unclear expressiveness
- explosion of the search space [APEX: 272x avg, up to 2,000x]
Goals and Challenges

Let’s raise the bar: full automation for advanced coverage criteria

- **specify** the coverage objective (+ unified treatment)
- **measure** coverage of test suites
- **cover** the objectives in an efficient manner (DSE)
- **unmask** the infeasible or redundant objectives
Let’s raise the bar: full automation for advanced coverage criteria

- **specify** the coverage objective (+ unified treatment)
  - labels, a simple specification mechanism
- **measure** coverage of test suites
  - thx to labels
- **cover** the objectives in an efficient manner (DSE)
  - DSE*, a variation of DSE
- **unmask** the infeasible or redundant objectives
  - an original combination of existing static analyses
The LTest plugin

LTest: All-in-one automated testing toolkit for C

- plugin of the FRAMA-C verification platform (open-source)
- based on PathCrawler for test generation
- the plugin itself is open-source except test generation
The LTest plugin

Supported criteria
- DC, CC, MCC, GACC
- FC, IDC, WM

- managed in a unified way
- rather easy to add new ones
The LTest plugin

Supported criteria
- DC, CC, MCC, GACC
- FC, IDC, WM

Managed in a unified way
- Rather easy to add new ones
Specify: Labels

- Annotate programs with **labels**
  - predicate attached to a specific program instruction

- Label \((loc, \varphi)\) is covered if a test execution
  - reaches the instruction at \(loc\)
  - satisfies the predicate \(\varphi\)

- **Good for us**
  - can easily encode a large class of coverage criteria
  - in the scope of standard program analysis techniques
Simulation of standard coverage criteria

Condition Coverage (CC)

```c
statement_1;
if (x==y && a<b) {...};
statement_3;
```

```c
statement_1;
// l1: x==y
// l2: !(x==y)
// l3: a<b
// l4: !(a<b)
if (x==y && a<b) {...};
statement_3;
```
Simulation of standard coverage criteria

Multiple-Condition Coverage (MCC)

```c
statement_1;
if (x==y && a<b) {...};
statement_3;
```

```c
statement_1;
// l1: x==y && a<b
// l2: x==y && a>=b
// l3: x!=y && a<b
// l4: x!=y && a>=b
if (x==y && a<b) {...};
statement_3;
```
And also Weak Mutations (WM)

Program P

\[
\text{statement i-1;}
\]
\[
x := y + z;
\]
\[
\text{statement i+1;}
\]

Output P(t)

Mutant M

\[
\text{statement i-1;}
\]
\[
x := y \times z;
\]
\[
\text{statement i+1;}
\]

Output M(t)

weak mutation

strong mutation
And also Weak Mutations (WM)

To simulate by labels, insert before the mutated instruction:

```c
// 11: y+z != y*z
```

**Diagram:**

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

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

- **Output P(t)**
  - `?`
  - `strong mutation`

- **Output M(t)**
  - `?`
  - `weak mutation`
To simulate by labels, insert before the mutated instruction:

```c
// l1: y+z != y*z
```

Out of scope:
- strong mutations, MCDC, def-use
- (side-effect weak mutations)
Covering label $l \iff$ Covering branch $\text{True}$

- ✓ sound & complete instrumentation
- × complexification of the search space [#paths, shape of paths]
- × dramatic overhead [theory & practice] [Apex: avg 272x, max 2000x]
Direct instrumentation is not good enough

Non-tightness 1

× P' has exponentially more paths than P

Direct instrumentation

\[2^N \text{ paths}\]
DSE* : Tight Instrumentation

Covering label 1 ⇔ Covering exit(0)

✓ sound & complete instrumentation
✓ no complexification of the search space
DSE* : Tight Instrumentation (2)

Direct instrumentation

1
\[ \Rightarrow \]
2
\[ p_1 \]

2^N paths

2
\[ \Rightarrow \]
pN

N

True
False

Tight Instrumentation

1
\[ \Rightarrow \]
2
\[ \text{non}_{-}\text{det} \]

N+1 paths

2
\[ \Rightarrow \]
N
\[ \text{non}_{-}\text{det} \]

\[ \text{asser}(p_1) \]

\[ \text{assert}(p_N) \]
Experiments

**Benchmark**: Standard benchmarks [*Siemens, Verisec, Mediabench*]
- 12 programs (50-300 loc), 3 criteria (*CC, MCC, WM*)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements

**Performance overhead**

<table>
<thead>
<tr>
<th></th>
<th>DSE</th>
<th>DSE’</th>
<th>DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>×1</td>
<td>×1.02</td>
<td>×0.49</td>
</tr>
<tr>
<td>Median</td>
<td>×1</td>
<td>×1.79</td>
<td>×1.37</td>
</tr>
<tr>
<td>Max</td>
<td>×1</td>
<td>×122.50</td>
<td>×7.15</td>
</tr>
<tr>
<td>Mean</td>
<td>×1</td>
<td>×20.29</td>
<td>×2.15</td>
</tr>
<tr>
<td>Timeouts</td>
<td>0</td>
<td>5 *</td>
<td>0</td>
</tr>
</tbody>
</table>

* : TO are discarded for overhead computation
cherry picking : 94s vs TO [1h30]
Experiments

Benchmark: Standard benchmarks [Siemens, Verisec, Mediabench]

- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements

Coverage

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>DSE</th>
<th>DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>37%</td>
<td>61%</td>
<td>62%</td>
</tr>
<tr>
<td>Median</td>
<td>63%</td>
<td>90%</td>
<td>95%</td>
</tr>
<tr>
<td>Max</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Mean</td>
<td>70%</td>
<td>87%</td>
<td>90%</td>
</tr>
</tbody>
</table>

vs DSE: +39% coverage on some examples
Experiments

Benchmark: Standard benchmarks [Siemens, Verisec, Mediabench]
- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements

DSE*
- DSE* significantly outperforms DSE’
- overhead kept reasonable
- better coverage than DSE
Automatic detection of uncoverable test objectives

- a *sound* method
- applicable to a large class of coverage criteria
- strong detection power, reasonable speed
- rely as much as possible on existing verification methods:

**Observation:**
Label \((loc, p)\) is *uncoverable* \(\iff\) Assertion \(\text{assert (} \neg p\text{);}\)

at location \(loc\) is *valid*
Unmask infeasibility

Automatic detection of uncoverable test objectives

- a sound method
- applicable to a large class of coverage criteria
- strong detection power, reasonable speed
- rely as much as possible on existing verification methods:

Observation:
Label \((\text{loc}, p)\) is uncoverable \(\iff\) Assertion \(\text{assert } (\neg p);\) at location \(\text{loc}\) is valid

Rely on a combination of
- abstract interpretation (infer context, not precise)
- weakest precondition (context-blind, locally precise)
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    // l1: res == 0
}
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {

    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0
}

VA and WP may fail
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x, a);
}

int g(int x, int a) {

    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0    // both VA and WP fail
}
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {
    //@assume 0 <= a <= 20
    //@assume 0 <= x <= 1000
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0
}
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x, a);
}

int g(int x, int a) {
    //@assume 0 <= a <= 20
    //@assume 0 <= x <= 1000
    int res;
    if(x + a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0      // VA ⊕ WP succeeds
Detection power

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

- 1,270 test requirements, **121 infeasible ones**

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1,270</td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>73</td>
<td>60%</td>
<td>118</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>29</td>
<td>100%</td>
<td>29</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>4.7</td>
<td>3.2</td>
<td>2.8</td>
<td>63%</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.8</td>
<td>82%</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>118</td>
<td>98%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>67%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>95%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels
Detection power

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

- 1,270 test requirements, **121 infeasible ones**

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#d</td>
<td>%d</td>
<td></td>
<td>%d</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1,270</td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>100%</td>
<td>29</td>
</tr>
<tr>
<td>Mean</td>
<td>4.7</td>
<td>3.2</td>
<td>63%</td>
<td>82%</td>
<td>4.5</td>
</tr>
</tbody>
</table>

#d : number of detected infeasible labels
%d : ratio of detected infeasible labels

- **VA ⊕ WP achieves almost perfect detection**
- detection speed is reasonable [$\leq 1s/obj.$]
Impact on test generation

report more accurate coverage ratio

<table>
<thead>
<tr>
<th>Detection method</th>
<th>None</th>
<th>VA ⊕ WP</th>
<th>Perfect*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>90.5%</td>
<td>99.2%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Min</td>
<td>61.54%</td>
<td>91.7%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Max</td>
<td>100.00%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Mean</td>
<td>91.10%</td>
<td>99.2%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

* preliminary, manual detection of infeasible labels
More recent work [Marcozzi et al. ICSE 2018]

- other sources of “pollution”:
  - duplicate and/or subsumed test objectives
  - harmful effect

- detection technique:
  - WP-based dedicated algorithms
  - enhanced with multi-core and fine tuning

- achievements:
  - detecting a large number of polluting test objectives (up to 27% of the total number of objectives)
  - scales: SQLite (200 kloc, 90k objectives, 9h, 15% identified as polluting)
Limitations of labels

- Labels encode **only criteria whose objectives are reachability constraints**
- Typical examples of criteria above labels:

**Call Coverage**
```c
int f() {
    if (...) { /* loc_1 */ g(); }
    if (...) { /* loc_2 */ g(); }
}
```

→ **cover loc_1 or loc_2**

**All-defs**
```c
/* loc_1 */ a := x;
if (...) /* loc_2 */ res := x+1;
else /* loc_3 */ res := x-1;
```

→ Cover **path loc_1 to loc_2**
  or **path loc_1 to loc_3**

**MCDC**
```c
statement_0;
// loc_1
if (x==y && a<b) {...};
statement_2;
```

→ Cover if condition **twice in a correlated way**:
  - a<b stays identical
  - x==y and (x==y && a<b) change

---

**Disjunction**  **Safety**  **Hyperproperties**
extend labels along the three axes (hyperlabels)

- $l \triangleright \{ v \mapsto \ldots \}$, $< h|\phi >$, $h \cdot h'$, $h + h'$, $h \rightarrow h'$

- give a formal semantic
- start extending LTest

- generic coverage measurement technique
- cover and unmask need update
Conclusion

**Dynamic Symbolic Execution is great!**
- ✓ robust, no false alarm, scale
- ✓ can be efficiently lifted to coverage-oriented testing

**Advanced test criteria can be fruitfully automated!**
- ✓ specify
- ✓ measure
- ✓ cover
- ✓ unmask