

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

Nikolai Kosmatov

joint work with Sébastien Bardin, Omar Chebaro, Mickaël Delahaye,
Michaël Marcozzi, Mike Papadakis, Virgile Prevosto. . .

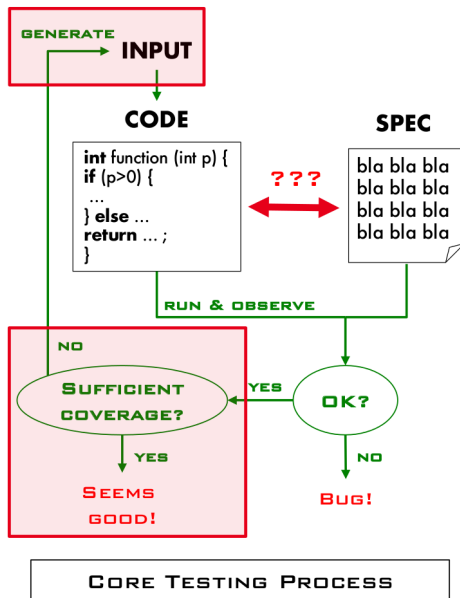
CEA, LIST, Software Safety and Security Lab
Paris-Saclay, France

Besançon, February 27th, 2018

Context: White-Box Testing

Testing process

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

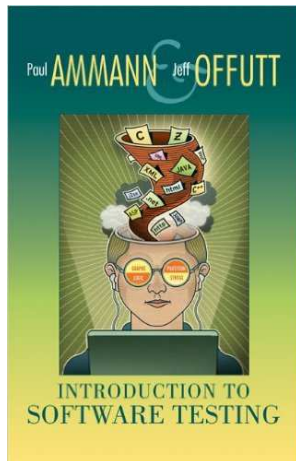


- **Framework:** white-box software testing process
- Automate test suite generation & coverage measure
- Coverage criterion = objectives to be fulfilled by the test suite
- Criterion guides automation
- Can be part of industrial normative requirements

Variety and sophistication gap between literature and testing tools

Literature:

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



Coverage criteria in white-box testing

Tools:

- Criteria seen as very dissimilar bases for automation
- Restricted to small subsets of criteria
- Extension is complex and costly

Tool name	BBC	FC	DC	CC	DCC	GACC	MCDC	MCC	BP	Other
Gcov	✓	✓	✓							0/19
Bullseye		✓			✓					0/19
Parasoft	✓	✓	✓	✓			✓		✓	0/19
Semantic Designs		✓	✓							0/19
Testwell CTC++	✓	✓			✓		✓			0/19

Global goal: bridge the gap between criteria and testing tools

Main ingredients of the talk:

Labels: a generic specification mechanism for coverage criteria

- ▶ can easily encode a large class of criteria
- ▶ a semantic view, with a formal treatment

DSE^{*}: an efficient test generation technique for labels

- ▶ an optimized version of DSE (Dynamic Symbolic Execution)
- ▶ no exponential blowup of the search space

LUncov: an efficient technique for detection of infeasible objectives

- ▶ based on existing static analysis techniques

LTest: an all-in-one testing toolset

- ▶ on top of FRAMA-C and PATHCRAWLER

HTOL: Hyperlabel Specification Language, extension of labels

- ▶ capable to encode almost all common criteria including MCDC

[Bardin et al., ICST 2014, TAP 2014, ICST 2015]

[Marcozzi et al., ICST 2017res, ICST 2017tool, ICSE 2018]

Main ingredients of the talk:

Labels: a generic specification mechanism for coverage criteria

- ▶ can easily encode a large class of criteria
- ▶ a semantic view, with a formal treatment

DSE^{*}: an efficient test generation technique for labels

- ▶ an optimized version of DSE (Dynamic Symbolic Execution)
- ▶ no exponential blowup of the search space

LUncov: an efficient technique for detection of infeasible objectives

- ▶ based on existing static analysis techniques

LTest: an all-in-one testing toolset

- ▶ on top of FRAMA-C and PATHCRAWLER

HTOL: Hyperlabel Specification Language, extension of labels

- ▶ capable to encode almost all common criteria including MCDC

Reminder: Goals

Specify and Measure, Cover and Unmask

Main ingredients of the talk:

Labels: a generic specification mechanism for coverage criteria

- ▶ can easily encode a large class of criteria
- ▶ a semantic view, with a formal treatment

Specify and Measure,

DSE*: an efficient test generation technique for labels

- ▶ an optimized version of DSE (Dynamic Symbolic Execution)
- ▶ no exponential blowup of the search space

Cover

LUncov: an efficient technique for detection of infeasible objectives

- ▶ based on existing static analysis techniques

and Unmask

LTest: an all-in-one testing toolset

- ▶ on top of FRAMA-C and PATHCRAWLER

HTOL: Hyperlabel Specification Language, extension of labels

- ▶ capable to encode almost all common criteria including MCDC

Reminder: Goals

Specify and Measure, Cover and Unmask

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

Basic definitions

Given a program P , a **label** l is a pair (loc, φ) , where:

- φ is a well-defined predicate at location loc in P
- φ contains no side-effects

Example:

```
statement_1;  
// l1:  x==y  
// l2:  !(x==y)  
if (x==y && a<b)  
    {...};  
statement_3;
```

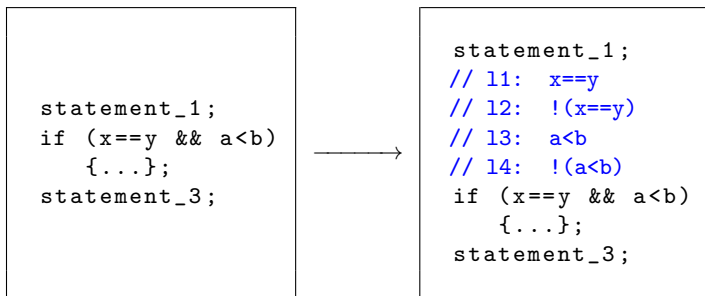
Basic definitions

- a test datum t **covers** l if $P(t)$ reaches loc and satisfies φ
- new criterion **LC** **label coverage**: requires to cover the labels

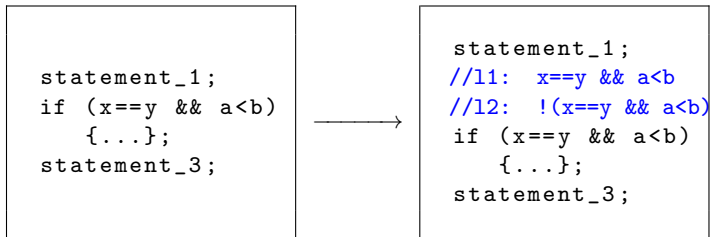
Example:

```
statement_1;  
// l1:  x==y  
// l2:  !(x==y)  
if (x==y && a<b)  
    {...};  
statement_3;
```

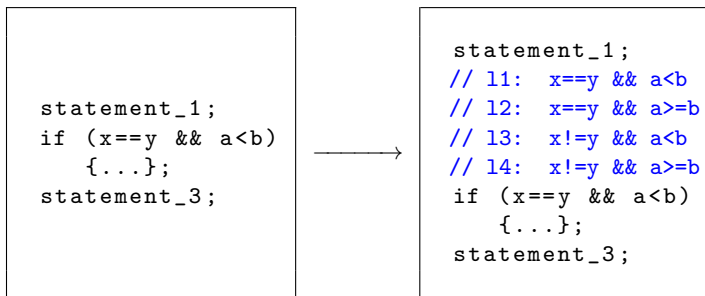
- a criterion **C** **can be simulated by LC** if for any P , after adding “appropriate” labels in P , TS covers **C** $\Leftrightarrow$ TS covers **LC**.



Condition Coverage (CC)

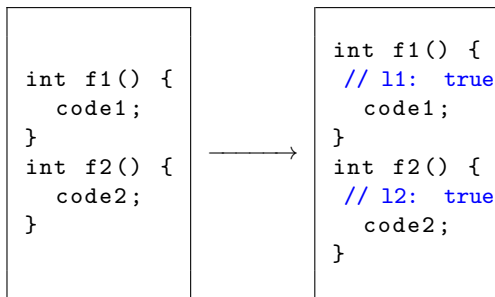


Decision Coverage (**DC**)



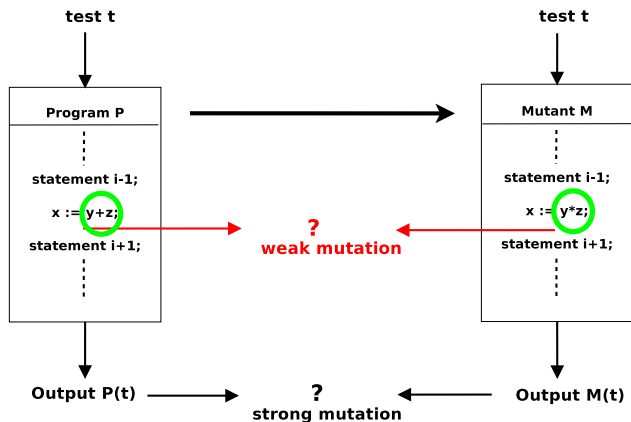
Multiple-Condition Coverage (**MCC**)

Simulation of coverage criteria by labels: FC



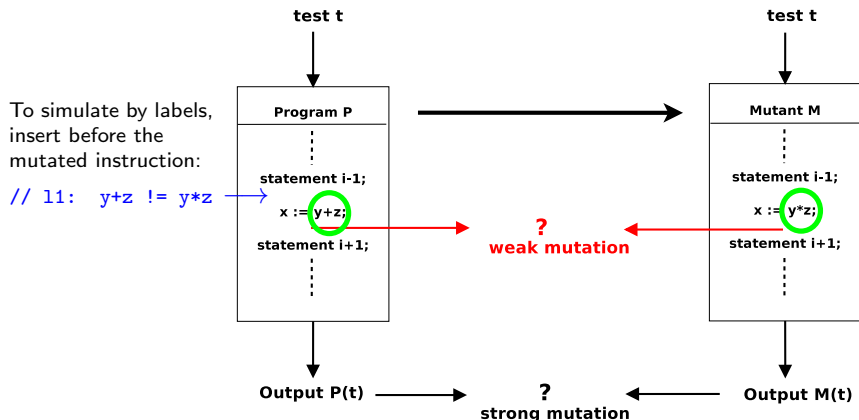
Function Coverage (**FC**)

Weak Mutation (WM) testing in a nutshell



- mutant M = syntactic modification of program P
- weakly covering M = finding t such that $P(t) \neq M(t)$ just after the mutation

Weak Mutation (WM) testing in a nutshell



- mutant M = syntactic modification of program P
- **weakly covering M** = finding t such that $P(t) \neq M(t)$ just after the mutation

Insert one label per mutant before the mutated instruction

Mutation inside a statement

- $lhs := e \quad \mapsto \quad lhs := e'$
 - ▶ insert label: $e \neq e'$
- $lhs := e \quad \mapsto \quad lhs' := e$
 - ▶ insert label: $\&lhs \neq \&lhs' \wedge (lhs \neq e \vee lhs' \neq e)$

Mutation inside a decision

- $\text{if (cond)} \quad \mapsto \quad \text{if (cond')}$
 - ▶ insert label: $cond \oplus cond'$

Beware: no side-effect inside labels

Theorem

*The following coverage criteria can be simulated by **LC**: **IC**, **DC**, **FC**, **CC**, **MCC**, Input Domain Partition, Run-Time Errors.*

Theorem

*For any finite set O of side-effect free mutation operators, **WM** _{O} can be simulated by **LC**.*

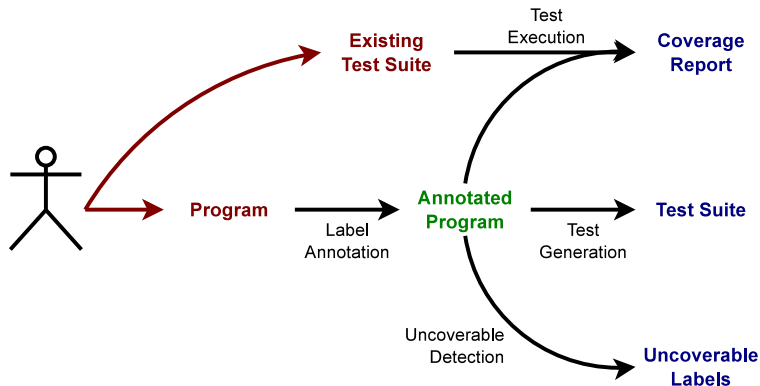
- Labels already enjoy a simple and efficient algorithm for coverage measurement
- Given a test suite TS and a program P
 - ▶ instrument P with checks for labels (P')
 - ▶ run every $t \in TS$ on P' , record covered labels
 - ▶ time cost: $\leq |TS| \cdot \max_{t \in TS}(P'(t))$
- **Works also for weak mutations**, whereas the standard algorithm for strong mutations is more costly:
 - ▶ create the set of mutants M
 - ▶ time cost: $\leq |TS| \cdot |M| \cdot \max_{m \in M, t \in TS}(m(t))$

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

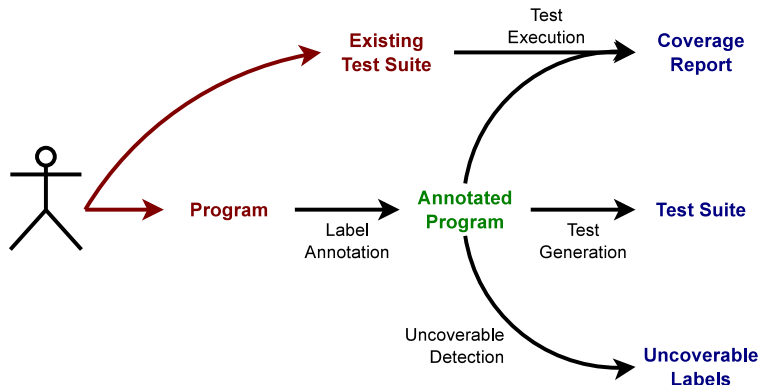
LTest is implemented on top of FRAMA-C

- FRAMA-C is a toolset for analysis of C programs
 - ▶ an extensible, open-source, plugin-oriented platform
 - ▶ offers value analysis (VA), weakest precondition (WP), specification language ACSL,...
- LTEST is open-source except test generation
 - ▶ based on the PATHCRAWLER test generation tool

The LTEST toolset for labels



The LTEST toolset for labels



A large set of supported criteria

- all treated in a unified way
- rather easy to add new ones

A typical use case

*// Checks if input points (x1,y1) and (x2,y2) lie
// in the same quadrant of the plane. Returns the
// quadrant number if so, otherwise returns 0.*

```
int quadrant (int x1, int y1, int x2, int y2){  
    if(x1 >= 0 && x2 >= 0 && y1 >= 0 && y2 >= 0)  
        return 1;    // (+,+): quadrant 1  
    if(x1 <= 0 && x2 <= 0 && y1 >= 0 && y2 >= 0)  
        return 2;    // (-,+): quadrant 2  
    if(x1 <= 0 && x2 <= 0 && y1 <= 0 && y2 <= 0)  
        return 3;    // (-,-): quadrant 3  
    if(x1 >= 0 && x2 >= 0 && y1 <= 0 && y2 <= 0)  
        return 4;    // (+,-): quadrant 4  
    return 0;        // not in the same quadrant  
}
```

Step 1: Specify test objectives for MCC

LTest automatically encodes test objectives by labels

Example. For the 3rd conditional (quadrant 3), 16 labels are inserted:

$$\begin{array}{l} x1 \leq 0 \wedge x2 \leq 0 \wedge y1 \leq 0 \wedge y2 \leq 0 \\ x1 \leq 0 \wedge x2 \leq 0 \wedge y1 \leq 0 \wedge y2 > 0 \\ x1 \leq 0 \wedge x2 \leq 0 \wedge y1 > 0 \wedge y2 \leq 0 \\ \vdots \\ x1 > 0 \wedge x2 > 0 \wedge y1 > 0 \wedge y2 \leq 0 \\ x1 > 0 \wedge x2 > 0 \wedge y1 > 0 \wedge y2 > 0 \end{array}$$

Result: Number of generated labels

16 labels generated for each conditional = **64 labels in total**

Reminder: Goals

Specify [✓] and Measure [], Cover [] and Unmask []

Step 1: Specify test objectives for MCC

LTest automatically encodes test objectives by labels

Example. For the 3rd conditional (quadrant 3), 16 labels are inserted:

$$\begin{array}{l} x1 \leq 0 \wedge x2 \leq 0 \wedge y1 \leq 0 \wedge y2 \leq 0 \\ x1 \leq 0 \wedge x2 \leq 0 \wedge y1 \leq 0 \wedge y2 > 0 \\ x1 \leq 0 \wedge x2 \leq 0 \wedge y1 > 0 \wedge y2 \leq 0 \\ \vdots \\ x1 > 0 \wedge x2 > 0 \wedge y1 > 0 \wedge y2 \leq 0 \\ x1 > 0 \wedge x2 > 0 \wedge y1 > 0 \wedge y2 > 0 \end{array}$$

Result: Number of generated labels

16 labels generated for each conditional = **64 labels in total**

Reminder: Goals

Specify [✓] and **Measure** [], **Cover** [] and **Unmask** []

Step 2: Measure the coverage of a test suite

LTest automatically measures test coverage

Example. For the test suite:

Test 1 : $x1 = 5$, $y1 = 8$, $x2 = 10$, $y2 = -15$

Test 2 : $x1 = 40$, $y1 = 15$, $x2 = -20$, $y2 = 26$

Result: Coverage ratio computed

8 labels are covered out of 64, thus MCC coverage ratio is 25%

Each test case is executed only once, and all covered test objectives are recorded

Reminder: Goals

Specify [✓] and Measure [✓], Cover [] and Unmask []

Step 2: Measure the coverage of a test suite

LTest automatically measures test coverage

Example. For the test suite:

Test 1 : $x1 = 5$, $y1 = 8$, $x2 = 10$, $y2 = -15$

Test 2 : $x1 = 40$, $y1 = 15$, $x2 = -20$, $y2 = 26$

Result: Coverage ratio computed

8 labels are covered out of 64, thus MCC coverage ratio is 25%

Each test case is executed only once, and all covered test objectives are recorded

Reminder: Goals

Specify [✓] and **Measure** [✓], Cover [] and Unmask []

Step 3: Generate test inputs to Cover MCC criterion

LTest automatically generates test inputs (using DSE^{*})

Results of DSE^{*} test generation

- Explores 409 program paths
- Generates a test suite that covers 58 labels out of 64

Reminder: Goals

Specify [✓] and Measure [✓], Cover [✓] and Unmask[]

What about the remaining 6 labels?

Are they really uncoverable?

If so, could they be detected before test generation?

Step 3: Generate test inputs to Cover MCC criterion

LTest automatically generates test inputs (using DSE^{*})

Results of DSE^{*} test generation

- Explores 409 program paths
- Generates a test suite that covers 58 labels out of 64

Reminder: Goals

Specify [✓] and Measure [✓], **Cover** [✓] and Unmask[]

What about the remaining 6 labels?

Are they really uncoverable?

If so, could they be detected before test generation?

Step 3: Generate test inputs to Cover MCC criterion

LTest automatically generates test inputs (using DSE^{*})

Results of DSE^{*} test generation

- Explores 409 program paths
- Generates a test suite that covers 58 labels out of 64

Reminder: Goals

Specify [✓] and Measure [✓], **Cover** [✓] and Unmask[]

What about the remaining 6 labels?

Are they really uncoverable?

If so, could they be detected before test generation?

Step 4: Unmask uncoverable labels

LTest automatically detects uncoverable labels (using LUncov)

Example of uncoverable label (2nd conditional)

```
if ( x1 >= 0 && x2 >= 0 && y1 >= 0 && y2 >= 0 )  
    return 1; // (+,+): quadrant 1  
// 128: x1 > 0  ^  x2 > 0  ^  y1 ≥ 0  ^  y2 ≥ 0  
if ( x1 <= 0 && x2 <= 0 && y1 >= 0 && y2 >= 0 )  
    return 2; // (-,+): quadrant 2
```

Results of detection with LUncov

6 labels are detected as **uncoverable** through static analysis

Benefits for test generation

- less paths to consider: here 284 paths instead of 409

Reminder: Goals

Specify [✓] and Measure [✓], Cover [✓] and Unmask[✓]

Step 4: Unmask uncoverable labels

LTest automatically detects uncoverable labels (using LUncov)

Example of uncoverable label (2nd conditional)

```
if ( x1 >= 0 && x2 >= 0 && y1 >= 0 && y2 >= 0 )  
    return 1; // (+,+): quadrant 1  
// 128: x1 > 0  ∧  x2 > 0  ∧  y1 ≥ 0  ∧  y2 ≥ 0  
if ( x1 <= 0 && x2 <= 0 && y1 >= 0 && y2 >= 0 )  
    return 2; // (-,+): quadrant 2
```

Results of detection with LUncov

6 labels are detected as **uncoverable** through static analysis

Benefits for test generation

- less paths to consider: here 284 paths instead of 409

Reminder: Goals

Specify [✓] and Measure [✓], Cover [✓] and Unmask[✓]

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

Dynamic Symbolic Execution [dart,cute,pathcrawler,exe,sage,pex,klée,...]

- ✓ very powerful approach to white-box test generation
- ✓ many tools and many successful case-studies since mid 2000's
- ✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]

Dynamic Symbolic Execution [dart,cute,pathcrawler,exe,sage,pex,klee,...]

- ✓ very powerful approach to white-box test generation
- ✓ many tools and many successful case-studies since mid 2000's
- ✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]

Symbolic Execution [King 70's]

- consider a program P on input v , and a given path σ
- a **path predicate** φ_σ for σ is a formula s.t. for any input v
 v satisfies $\varphi_\sigma \Leftrightarrow P(v)$ follows σ
- old idea, recently renewed interest [requires powerful solvers]

Dynamic Symbolic Execution [dart,cute,pathcrawler,exe,sage,pex,klee,...]

- ✓ very powerful approach to white-box test generation
- ✓ many tools and many successful case-studies since mid 2000's
- ✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]

Symbolic Execution [King 70's]

- consider a program P on input v , and a given path σ
- a **path predicate** φ_σ for σ is a formula s.t. for any input v
 v satisfies $\varphi_\sigma \Leftrightarrow P(v)$ follows σ
- old idea, recently renewed interest [requires powerful solvers]

Dynamic Symbolic Execution [Korel+, Williams+, Godefroid+]

- interleaves dynamic and symbolic executions
- drives the search towards feasible paths for free
- gives hints for relevant under-approximations

Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_σ satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]

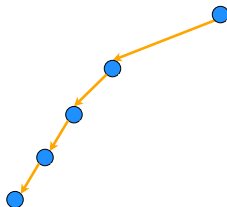
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_σ satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



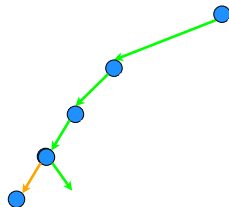
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_{σ} satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



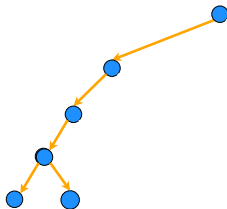
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_{σ} satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



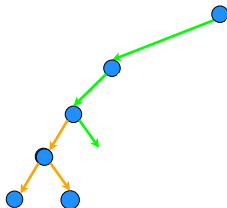
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_σ satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



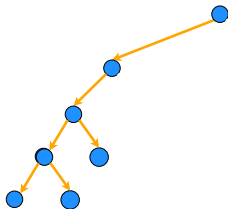
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_σ satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



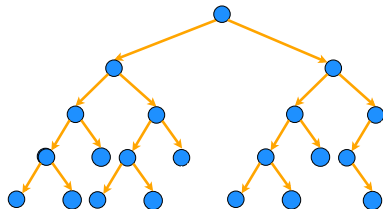
Dynamic Symbolic Execution (2)

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(P)$

- pick an uncovered path $\sigma \in Paths^{\leq k}(P)$
- is the path predicate φ_σ satisfiable?
- if SAT(s) then add a new pair $\langle s, \sigma \rangle$ into TS
- loop until no more paths to cover

[smt solver]



Dynamic Symbolic Execution

- ✓ very powerful approach to white-box test generation
- ✓ arguably one of the most wide-spread use of formal methods in “common software”

Dynamic Symbolic Execution

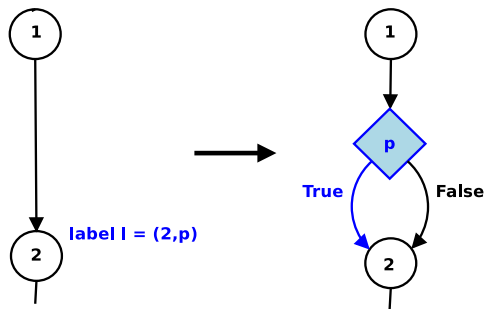
- ✓ very powerful approach to white-box test generation
 - ✓ arguably one of the most wide-spread use of formal methods in “common software”
 - ✗ lack of support for many coverage criteria
-

Dynamic Symbolic Execution

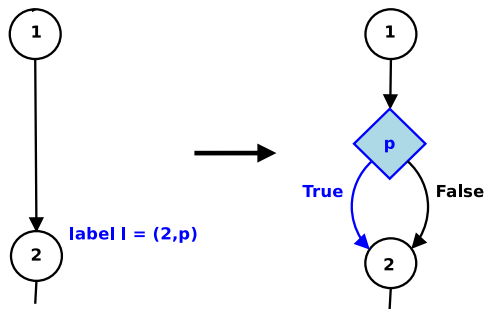
- ✓ very powerful approach to white-box test generation
- ✓ arguably one of the most wide-spread use of formal methods in “common software”
- ✗ lack of support for many coverage criteria

Challenge: extend DSE to a large class of coverage criteria

- well-known problem
- recent efforts in this direction through instrumentation
[Active Testing, Mutation DSE, Augmented DSE]
- limitations:
 - ▶ exponential explosion of the search space [AP_{EX}: 272x avg]
 - ▶ very implementation-centric mechanisms
 - ▶ unclear expressiveness



Covering label $I \Leftrightarrow$ Covering branch True

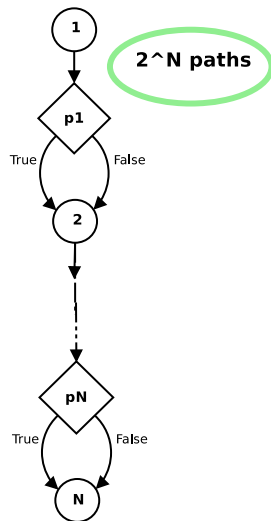


Covering label $I \Leftrightarrow$ Covering branch True

✓ sound & complete instrumentation w.r.t. **LC**

Direct instrumentation P' is not good enough

Direct instrumentation

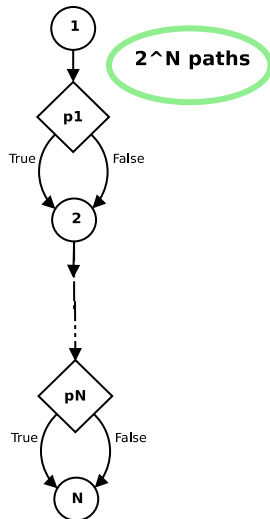


Direct instrumentation P' is not good enough

Non-tightness 1

✗ P' has exponentially more paths than P

Direct instrumentation



Direct instrumentation P' is not good enough

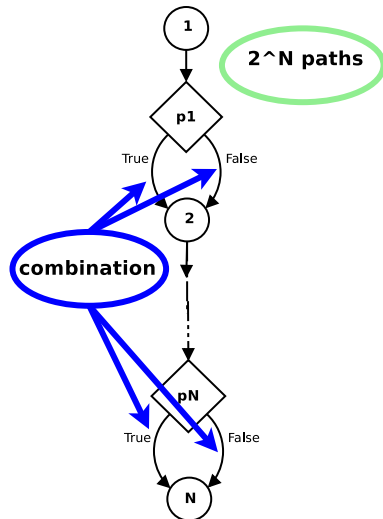
Non-tightness 1

- ✗ P' has exponentially more paths than P

Non-tightness 2

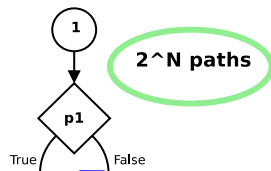
- ✗ Paths in P' too complex
 - ▶ at each label, require to cover p or to cover $\neg p$
 - ▶ π' covers up to N labels

Direct instrumentation

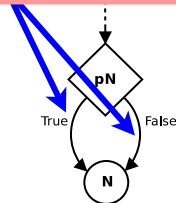


Direct instrumentation P' is not good enough

Direct instrumentation

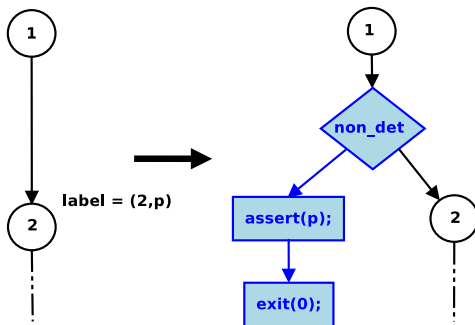


- ✓ sound & complete instrumentation w.r.t. **LC**
- ✗ dramatic overhead [theory & practice]

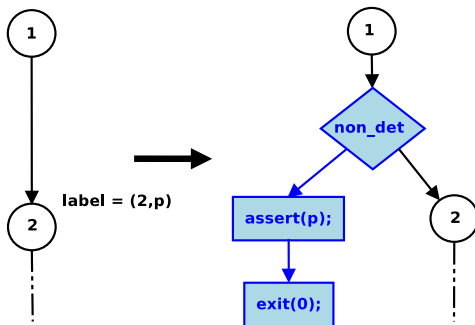


The DSE^{*} algorithm

- Tight instrumentation P^* : totally prevents “complexification”
- Iterative Label Deletion: discards some redundant paths
- Both techniques can be implemented in a black-box manner



Covering label `l` $\Leftrightarrow$ Covering `exit(0)`

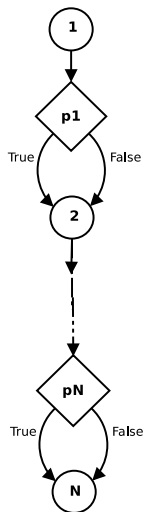


Covering label $l \Leftrightarrow$ Covering `exit(0)`

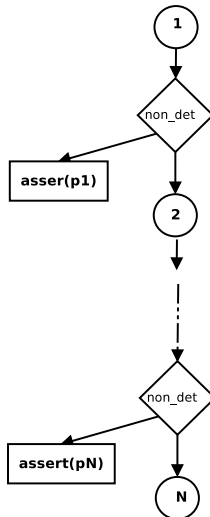
✓ sound & complete instrumentation w.r.t. LC

DSE*: Direct vs tight instrumentation, P' vs P^*

Direct instrumentation

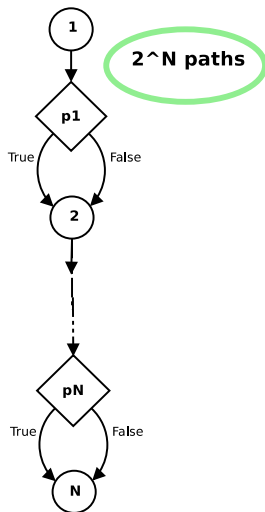


Tight Instrumentation

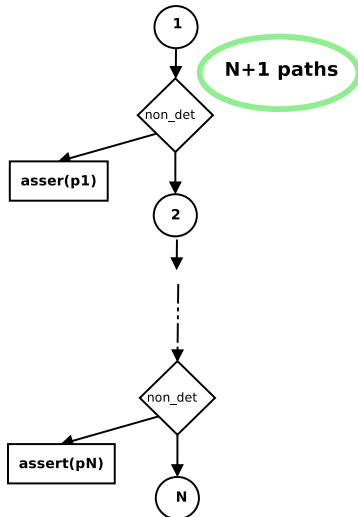


DSE*: Direct vs tight instrumentation, P' vs P^*

Direct instrumentation

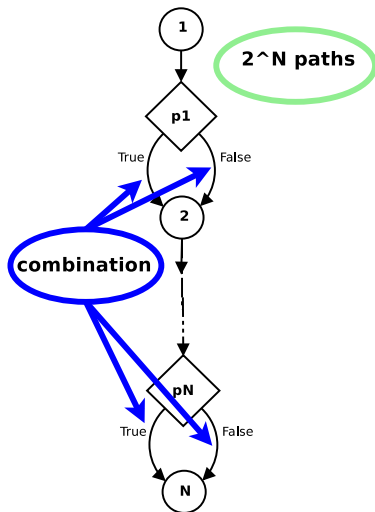


Tight Instrumentation

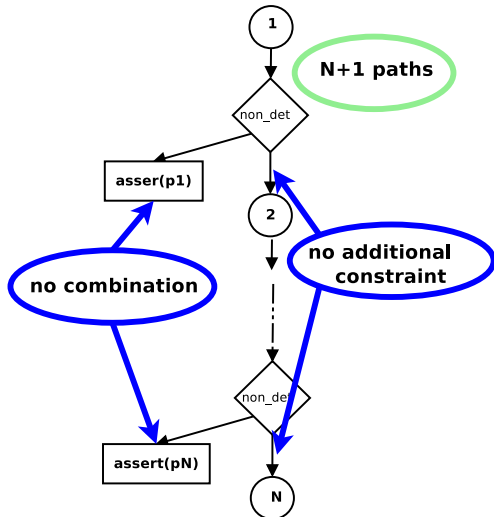


DSE*: Direct vs tight instrumentation, P' vs P^*

Direct instrumentation

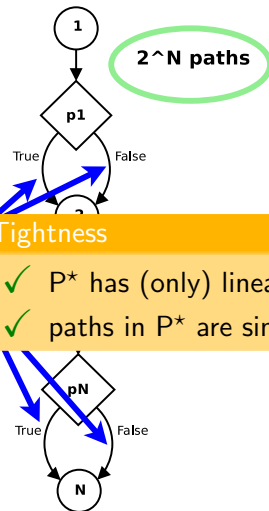


Tight Instrumentation

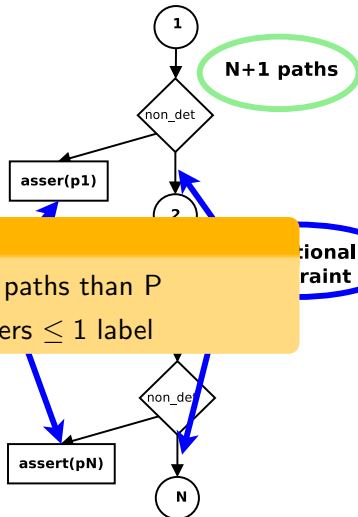


DSE*: Direct vs tight instrumentation, P' vs P^*

Direct instrumentation



Tight Instrumentation

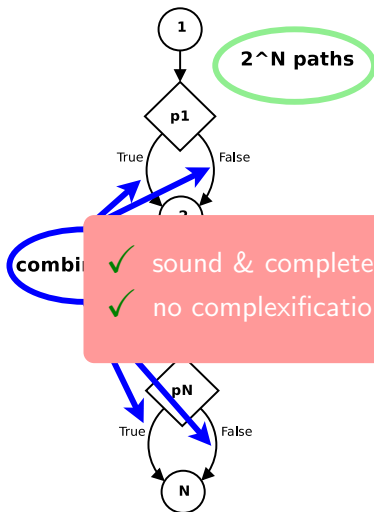


Tightness

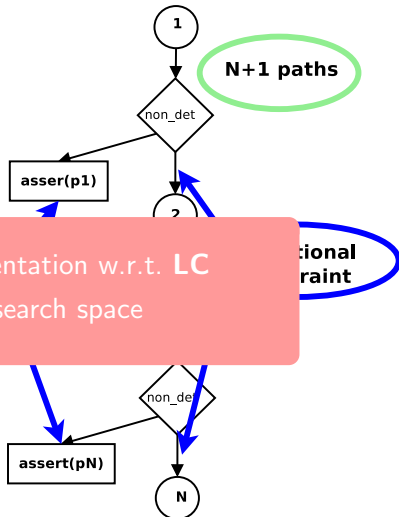
- ✓ P^* has (only) linearly more paths than P
- ✓ paths in P^* are simple: covers ≤ 1 label

DSE*: Direct vs tight instrumentation, P' vs P^*

Direct instrumentation



Tight Instrumentation



- ✓ sound & complete instrumentation w.r.t. LC
- ✓ no complexification of the search space

Observations

- we need to cover each label only once
- yet, DSE explores paths of P^* ending in already-covered labels
- we burden DSE with “useless” paths w.r.t. **LC**

Observations

- we need to cover each label only once
- yet, DSE explores paths of P^* ending in already-covered labels
- we burden DSE with “useless” paths w.r.t. **LC**

Solution: Iterative Label Deletion

- keep a *covered/uncovered* status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates the status [truly requires DSE]

Implementation

- symbolic part: a slight modification of P^*
- dynamic part: a slight modification of P'

Observations

- we need to cover each label only once
- yet, DSE explores paths of P^* ending in already-covered labels
- we burden DSE with “useless” paths w.r.t. **LC**

Solution: Iterative Label Deletion

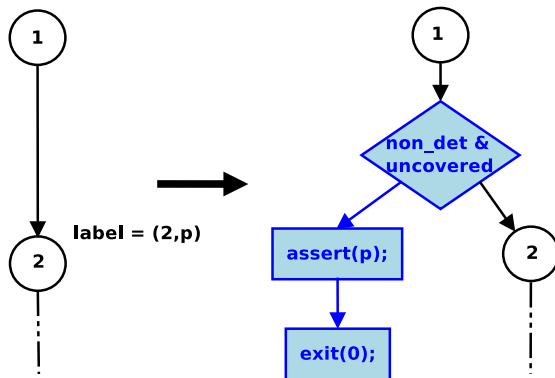
- keep a *covered/uncovered* status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates the status [truly requires DSE]

Implementation

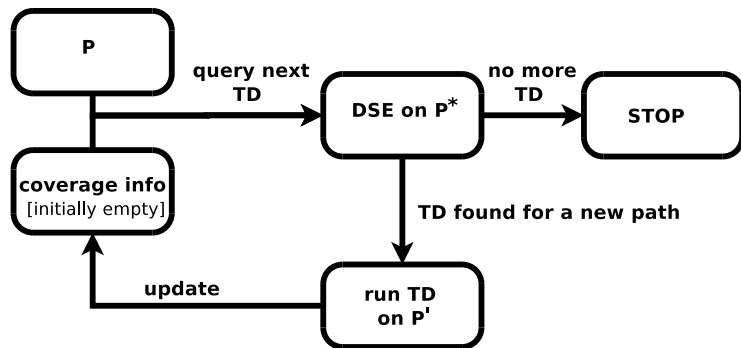
- symbolic part: a slight modification of P^*
- dynamic part: a slight modification of P'

Iterative Label Deletion is relatively complete w.r.t. **LC**

DSE*: Iterative Label Deletion (2)



DSE*: Iterative Label Deletion (3)



The DSE^{*} algorithm

- Tight instrumentation P^* : totally prevents “complexification”
- Iterative Label Deletion: discards some redundant paths
- Both techniques can be implemented in black-box

Implementation

- inside PATHCRAWLER
- follows DSE*
- search heuristics: “label-first DFS”
- run in deterministic mode

Goal of experiments

- evaluate DSE* versus DSE'
- evaluate overhead of handling labels

Benchmark programs

- SQLite, OpenSSL
- 12 programs taken from standard DSE benchmarks (Siemens, Verisec, MediaBench)
- 3 coverage criteria: **CC**, **MCC**, **WM**

Results

- DSE': 4 timeouts (TO), max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average: 2.4x)
- on one example, 94s instead of a TO [1h30]
- DSE* achieves very high **LC**-coverage [$> 90\%$ on 28/36]
- after a static analysis step for detection of uncoverable labels, it becomes even higher [$> 99\%$]

Results

- DSE': 4 timeouts (TO), max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average: 2.4x)
- on one example, 94s instead of a TO [1h30]
- DSE* achieves very high LC-coverage [$> 90\%$ on 28/36]
- after a static analysis step for detection of uncoverable labels, it becomes even higher [$> 99\%$]

Results

- DSE': 4 timeouts (TO), max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average: 2.4x)
- on one example, 94s instead of a TO [1h30]
- DSE* achieves very high **LC**-coverage [$> 90\%$ on 28/36]
- after a static analysis step for detection of uncoverable labels, it becomes even higher [$> 99\%$]

Results

- DSE': 4 timeouts (TO), max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average: 2.4x)
- on one example, 94s instead of a TO [1h30]
- DSE* achieves very high **LC**-coverage [$> 90\%$ on 28/36]
- after a static analysis step for detection of uncoverable labels, it becomes even higher [$> 99\%$]

Conclusion

- DSE* performs significantly better than DSE'
- The overhead of handling labels is kept reasonable
- still room for improvement

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

The 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)

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 $\Leftrightarrow$ Assertion `assert ( $\neg p$ );`
at location loc is valid

■ **Forward abstract interpretation, or Value Analysis (VA)**

[state approximation]

- ▶ compute an invariant of the program
- ▶ then, analyze all assertions (labels) in one run
- ▶ global but limited reasoning

■ **Weakest precondition calculus (WP)** [goal-oriented]

- ▶ perform a dedicated check for each assertion
- ▶ a single check usually easier, but many of them
- ▶ local but precise reasoning

Example: program with two uncoverable labels

```
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;    // the only possible outcome
    else
        res = 0;
    // l1: res == 0
    // l2: res == 2
    return res;
}
```

Example: program with two valid assertions

```
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;    // the only possible outcome
    else
        res = 0;
    //@ assert res != 0
    //@ assert res != 2
    return res;
}
```

Example: program with two valid assertions

```
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;    // the only possible outcome
    else
        res = 0;
    //@ assert res != 0    // both VA and WP fail
    //@ assert res != 2    // detected as valid
    return res;
}
```


Goal: get the best of the two worlds

- Idea: VA passes to WP the global information that WP needs

Which information, and how to transfer it?

- VA computes variable domains
- WP naturally takes into account assumptions (`assume`)

Proposed solution:

- **VA exports computed variable domains in the form of WP-assumptions**

Example: alone, 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) {

    int res;
    if(x+a >= x)
        res = 1;    // the only possible outcome
    else
        res = 0;
    //@ assert res != 0    // both VA and WP fail

    return res;
}
```

Example: combination $VA \oplus WP$ succeeds

```
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 // VA inserts domains...
    int res;
    if(x+a >= x)
        res = 1;    // the only possible outcome
    else
        res = 0;
    //@ assert res != 0

    return res;
}
```

Example: combination $VA \oplus WP$ succeeds

```
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 // VA inserts domains...
    int res;
    if(x+a >= x)
        res = 1;    // the only possible outcome
    else
        res = 0;
    //@ assert res != 0    // ... and WP succeeds!

    return res;
}
```

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

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

	#Lab	#Inf	VA		WP		VA $\oplus$ WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	60%	118	98%
Min		0	0	0%	0	0%	2	67%
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	82%	4.5	95%

#d: number of detected infeasible labels

%d: ratio of detected infeasible labels

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

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

	#Lab	#Inf	VA		WP		VA $\oplus$ WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	60%	118	98%
Min		0	0	0%	0	0%	2	67%
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	82%	4.5	95%

#d: number of detected infeasible labels

%d: ratio of detected infeasible labels

- VA $\oplus$ WP achieves **almost perfect detection**
- detection speed is reasonable [$\leq 1s/obj.$]

report more accurate coverage ratio

Detection method	Coverage ratio reported by DSE*		
	None	VA $\oplus$ WP	Perfect*
Total	90.5%	99.2%	100.0%
Min	61.54%	91.7%	100.0%
Max	100.00%	100.0%	100.0%
Mean	91.10%	99.2%	100.0%

* preliminary, manual detection of infeasible labels

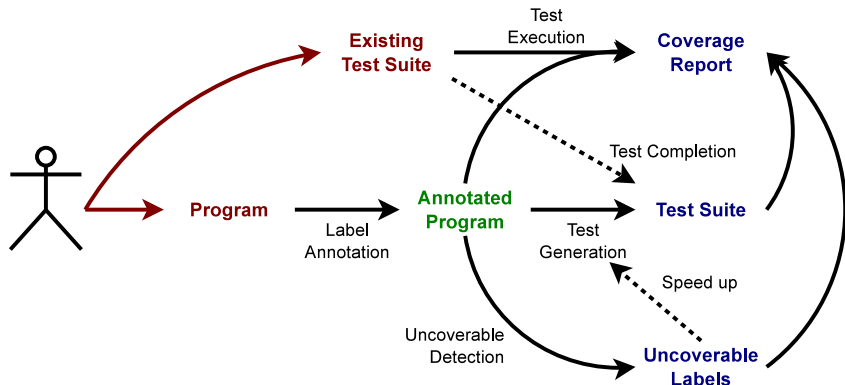
- automatic, sound and generic method
- new combination of existing verification techniques
- experiments for 12 programs and 3 criteria (CC, MCC, WM):
 - ▶ strong detection power (95%),
 - ▶ reasonable detection speed ($\leq 1\text{s}/\text{obj.}$),
 - ▶ test generation speedup (3.8x in average),
 - ▶ more accurate coverage ratios (99.2% instead of 91.1% in average, 91.6% instead of 61.5% minimum)

[Bardin et al. ICST 2014, TAP 2014, ICST 2015]

Most recent work [Marcozzi et al. ICSE 2018]

- other sources of “pollution”:
 - ▶ duplicate and/or subsumed test objectives
 - ▶ harmful effect [Papadakis et al., ISSTA 2016]
- 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: OpenSSL, gzip, SQLite

LUncov in the LTEST toolset for labels



Uses static analyzers from FRAMA-C

- sound detection of uncoverable labels

Service cooperation

- share label statuses
- Covered, Infeasible, ?

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

MCDC:

- coverage criterion used in aeronautics
- demanding, industrially relevant
- requires to show that any atomic condition c_i in decision $D = D(c_1, \dots, c_n)$ can independently influence the decision D
 - ▶ there exist two tests t_1 and t_2 such that:
 - ▶ $\forall j \neq i$, values of c_j on t_1 and t_2 are the same
 - ▶ values of c_i on t_1 and t_2 are different
 - ▶ values of D on t_1 and t_2 are different

Example: show that a alone can influence $D = a \wedge (b \vee c)$

- $t_1: (a = 1, b = 1, c = 0), D = 1$
- $t_2: (a = 0, b = 1, c = 0), D = 0$

GACC (global active clause coverage, a.k.a. shortcut MCDC)

- a weaker interpretation of MCDC, still demanding and industrially relevant
- the notion of "influences alone" is different
 - ▶ there exist two tests t_1 and t_2 such that:
 - ▶ $c_i = \mathbf{False}$ on t_1
 - ▶ $c_i = \mathbf{True}$ on t_2
 - ▶ for both t_1 and t_2 , modifying c_i alone switches D
 - ▶ [no explicit link between t_1 and t_2 for the values of c_j or D]

Example: show that a alone can influence decision $D = a \wedge (b \vee c)$

- $t_1: (a = 1, b = 1, c = 0), D = 1$
- $t_2: (a = 0, b = 1, c = 1), D = 1$
- OK for GACC, not for MCDC

GACC can be encoded into labels [Tillmann et al. 2010]

- [no exact encoding is known for MCDC]

$$\begin{aligned}\varphi_i &\triangleq c_i = \mathbf{F} \wedge D(c_1, \dots, c_{i-1}, \mathbf{T}, c_{i+1}, \dots, c_n) \neq \\ &\quad C(c_1, \dots, c_{i-1}, \mathbf{F}, c_{i+1}, \dots, c_n) \\ \varphi'_i &\triangleq c_i = \mathbf{T} \wedge D(c_1, \dots, c_{i-1}, \mathbf{T}, c_{i+1}, \dots, c_n) \neq \\ &\quad C(c_1, \dots, c_{i-1}, \mathbf{F}, c_{i+1}, \dots, c_n)\end{aligned}$$

where **F** = **False**, **T** = **True**

Limitations of labels

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

Call Coverage

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

→ cover loc_1 **or** loc_2

All-defs

```
/* 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

```
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

- A formal extension adding 5 operators to combine labels together (**hyperlabels**):

$l ::= \ell \triangleright B$ atomic label with bindings

$B ::= \{v_1 \leftarrow e_1; \dots\}$ bindings

$h ::= l$ label

	$[l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i}\}^* l_n]$	sequence of labels
	$\langle h \mid \psi \rangle$	guarded hyperlabel
	$h_1 \cdot h_2$	conjunction of hyperlabels
	$h_1 + h_2$	disjunction of hyperlabels

Formal Semantics:

LABEL $\frac{t \in TS \quad t \rightsquigarrow_P^k \langle loc, s \rangle \quad s \models \varphi \quad \mathcal{E} \supseteq \llbracket B \rrbracket_s}{t \rightsquigarrow_{\mathcal{E}}^k \langle loc, \varphi \rangle \triangleright B \quad \langle TS, \mathcal{E} \rangle \rightsquigarrow_P \langle loc, \varphi \rangle \triangleright B}$		GUARD $\frac{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h \quad \mathcal{E} \models \psi}{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P \langle h \mid \psi \rangle}$	CONJUNCTION $\frac{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 \quad \langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_2}{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 \cdot h_2}$
DISJUNCTION LEFT $\frac{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1}{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 + h_2}$	DISJUNCTION RIGHT $\frac{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_2}{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 + h_2}$	SEQUENCE $\frac{t \in TS \quad \forall i \in [1, n], t \rightsquigarrow_{\mathcal{E}}^{k_i} l_i \quad \forall i \in [1, n-1], k_i < k_{i+1} \quad \forall i \in [1, n-1], \forall j \in]k_i, k_{i+1}[, (loc_j, s_j) = P(t)_j \wedge \phi_i(\mathcal{E}, loc_j, s_j)}{\langle TS, \mathcal{E} \rangle \rightsquigarrow_P [l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i}\}^* l_n]}$	

Naming convention: TS test suite; $\mathcal{E}$ hyperlabel environment; h, h_1, h_2 hyperlabels; ψ hyperlabel guard predicate; n positive integer; $l_1, \dots, l_n$ atomic labels with bindings; t test datum; $k, k_1, \dots, k_n$ execution step numbers; loc_j, loc program locations; s_j, s execution states; $P(t)_j$ the j -th step of the program run $P(t)$ of P on t ; $\phi_1, \dots, \phi_n$ predicates over sequences of labels; φ label predicate; B hyperlabel bindings.

- Hyperlabels add operators to combine labels together!

Call Coverage

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

→ cover loc_1 or loc_2



$(loc_1, true) + (loc_2, true)$

All-defs

```
/* 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

```
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 d=(x==y && a<b)
change

- Hyperlabels add operators to combine labels together!

Call Coverage

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

→ cover *loc_1* or *loc_2*

All-defs

```
/* 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*


$$((loc_1, true) \rightarrow (loc_2, true)) + ((loc_1, true) \rightarrow (loc_3, true))$$

MCDC

```
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 $d = (x == y \ \&\& \ a < b)$ change

- Hyperlabels add operators to combine labels together!

Call Coverage

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

→ cover loc_1 **or** loc_2

All-defs

```
/* 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

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

→ Cover if condition **twice**
in a correlated way:

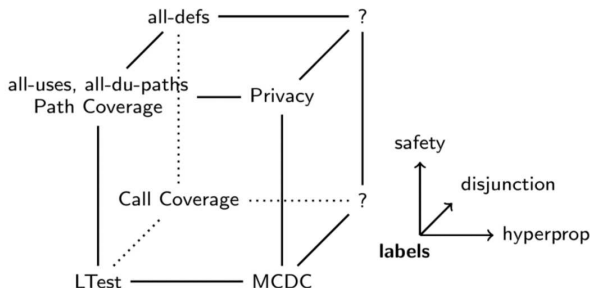
- a<b stays identical
- x==y and d=(x==y && a<b)
change



$$\begin{aligned} l &\triangleq (loc_1, d) \triangleright \{c_1 \leftarrow x==y; c_2 \leftarrow a<b\} \\ l' &\triangleq (loc_1, \neg d) \triangleright \{c'_1 \leftarrow x==y; c'_2 \leftarrow a<b\} \\ h_1 &\triangleq \langle l \cdot l' \mid c_1 \neq c'_1 \wedge c_2 = c'_2 \rangle \end{aligned}$$

HTOL: Taxonomy of coverage criteria

- Labels can encode test objectives that are **reachability constraints**
- RESULT 1:** labels must be extended along **three orthogonal directions** to handle other criteria:



- **RESULT 2:** Formal definition of the **hyperlabel language** (HTOL)

- Extends labels into the three directions
- Adds support for **all criteria** including MCDC (except from full mutations)
- Offers nice other testing perspectives (e.g. security hyperproperties, like non-interference)

Tool name	BBC	FC	DC	CC	DCC	GACC	MCDC	MCC	BP	Other
Gcov	✓	✓	✓							0/19
Bullseye		✓			✓					0/19
Parasoft	✓	✓	✓	✓			✓		✓	0/19
Semantic Designs		✓	✓							0/19
Testwell CTC++	✓	✓			✓		✓			0/19
LTest	✓	✓	✓	✓	✓	✓		✓		4/19
Hyper-LTest	✓	✓	✓	✓	✓	✓	✓	✓	✓	18/19

- **RESULT 3:** Extension of **Ltest** to hyperlabels (in progress, essentially coverage)

→ Current work provides a full-featured testing tool for all criteria

(yet, test generation is suboptimal, since hyperlabels not considered)

- 1 Labels
- 2 LTest: an all-in-one testing toolset
- 3 Efficient test generation for labels
 - Dynamic Symbolic Execution (DSE)
 - DSE*: optimized test generation for labels
- 4 Detection of infeasible test objectives
- 5 Hyperlabel Specification Language (HTOL)
- 6 Conclusion

Summary

Labels: a generic specification mechanism for coverage criteria

- ▶ can easily encode a large class of criteria
- ▶ a semantic view, with a formal treatment

DSE*: an efficient test generation technique for labels

- ▶ an optimized version of DSE (Dynamic Symbolic Execution)
- ▶ no exponential blowup of the search space

LUncov: an efficient technique for detection of infeasible objectives

- ▶ based on existing static analysis techniques

LTest: an all-in-one testing toolset

- ▶ on top of FRAMA-C and PATHCRAWLER

HTOL: Hyperlabel Specification Language, extension of labels

- ▶ capable to encode almost all common criteria including MCDC

Reminder: Goals

Specify [✓] and Measure, [✓], Cover [✓] and Unmask [✓]

- An efficient dedicated support of hyperlabels in test generation (DSE)
- Further optimizations of LTest (e.g. detection of uncoverable hyperlabels)
- Developing the emerging interest for LTest in industry