Advanced Test Coverage Criteria: Specification and Support in Automatic Testing Tools

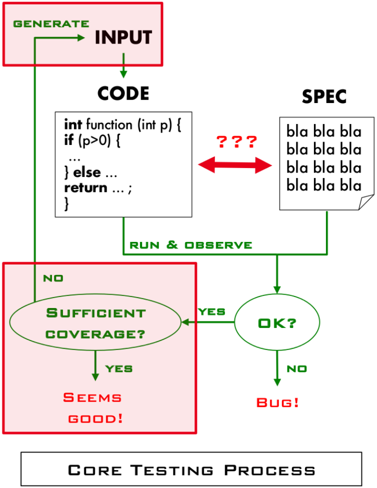
Nikolai Kosmatov

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

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

TAROT 2017, Napoli, June 26, 2017

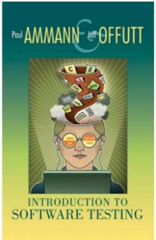
Context : White-Box Testing



- 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 industrial normative requirements

Coverage criteria in white-box testing

- Variety and sophistication gap between literature and testing tools
- <u>Literature</u>: 28 white-box criteria in the Ammann & Offutt book
- <u>Tools</u>: criteria seen as very dissimilar bases for automation
  - ➢ Restricted to small subset 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

- based on predicates, can easily encode a large class of criteria
- w.r.t related work : semantic view, more formal treatment

DSE* : an efficient integration of labels into DSE

- no exponential blowup of the search space
- can be added to DSE in a black-box manner

LTest : Implementation on top of FRAMA-C and PATHCRAWLER

- huge savings compared to existing approaches
- handles labels with a very low overhead (2x average, up to 7x)

HTOL : Hyperlabel Specification Language, extension of labels

- capable to encode almost all common criteria

[Bardin et al., ICST 2014, TAP 2014, ICST 2015]
[Marcozzi et al., ICST 2017 (research), ICST 2017 (tool)]

Outline

1. Dynamic Symbolic Execution (DSE)
2. Labels
   - Notation
   - Expressiveness
3. Efficient DSE for labels
   - Direct instrumentation
   - DSE*
   - Tight instrumentation
   - Iterative Label Deletion
4. LTest toolset : Implementation and Experiments
5. Hyperlabel Specification Language (HTOL)
6. Conclusion

Dynamic Symbolic Execution

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]

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)

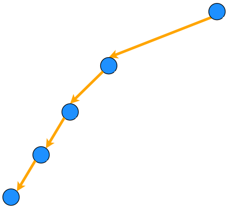
input : a program P
output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

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

Dynamic Symbolic Execution (2)

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

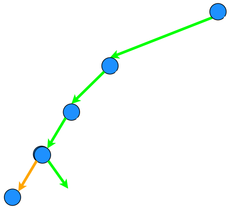
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable? [smt solver]
- if SAT(s) then add a new pair $<$ s, σ $>$ into TS
- loop until no more paths to cover



input : a program P

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

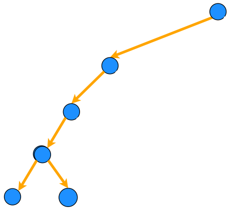
- pick an uncovered path $\sigma \in \mathit{Paths}^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $<$ s, σ $>$ into *TS*
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

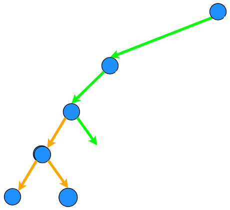
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $<$ s, σ $>$ into *TS*
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

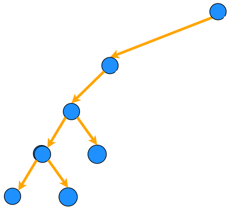
- pick an uncovered path $\sigma \in Paths^{\leq k}(\texttt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $< s, \sigma >$ into *TS*
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

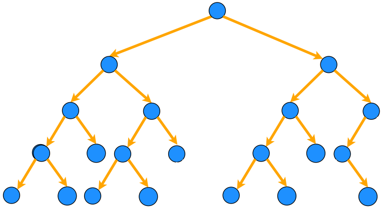
- pick an uncovered path $\sigma \in Paths^{\leq k}(\texttt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $<$ s, $\sigma >$ into TS
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

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



The problem

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”

The problem

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

The problem

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 [APEX : 272x avg]
  - very implementation-centric mechanisms
  - unclear expressiveness

Outline

Labels and the notion of simulation

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

- φ is a well-defined predicate in P at location *loc*
- φ contains no side-effect expression

Basic definitions

- a test datum t covers l if $P(t)$ reaches *loc* and satisfies φ
- new criterion **LC** (label coverage) for annotated programs
- 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**.

Goal : show the relative expressiveness of **LC**

Simulation of coverage criteria by labels : DC

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

→

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

Decision Coverage (**DC**)

Simulation of coverage criteria by labels : CC

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

→

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

Condition Coverage (**CC**)

Simulation of coverage criteria by labels : MCC

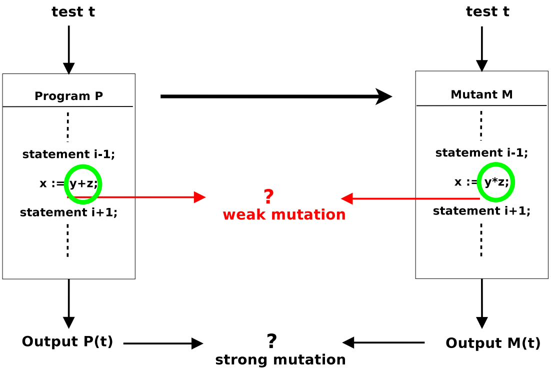
```
statement_1;
if (x==y && a<b)
   {...};
statement_3;
```

→

```
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;
```

Multiple-Condition Coverage (**MCC**)

Weak Mutation (WM) testing in a nutshell



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

Simulation of coverage criteria by labels : WM

One label per mutant

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`
  - add label : $e \neq e'$
- `lhs := e` $\mapsto$ `lhs' := e`
  - add label : $\&lhs \neq \&lhs' \wedge (lhs \neq e \vee lhs' \neq e)$

Mutation inside a decision

- `if (cond)` $\mapsto$ `if (cond')`
  - add label : $cond \oplus cond'$

Beware : no side-effect inside labels

Simulation results

Theorem

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

Theorem

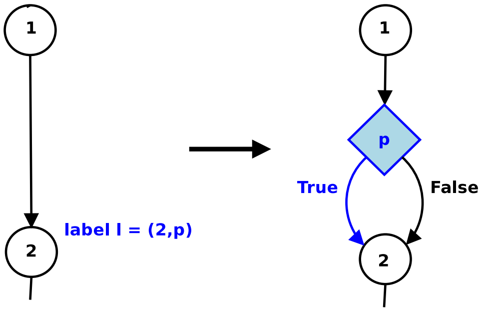
For any finite set O of side-effect free mutation operators, $\mathbf{WM}_O$ *can be simulated by* **LC**.

Goals

- ✓ GOAL1 : generic specification mechanism for coverage criteria
- ☐ GOAL2 : efficient integration into DSE

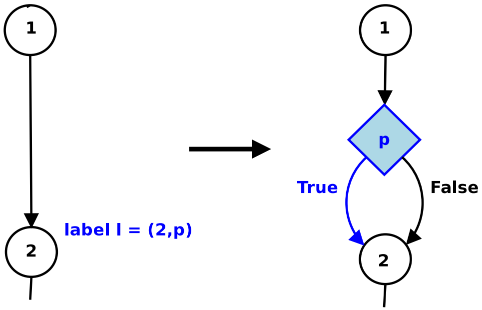
Outline

Direct instrumentation P' [APEX, Mutation DSE]



Covering label l $\Leftrightarrow$ Covering branch `True`

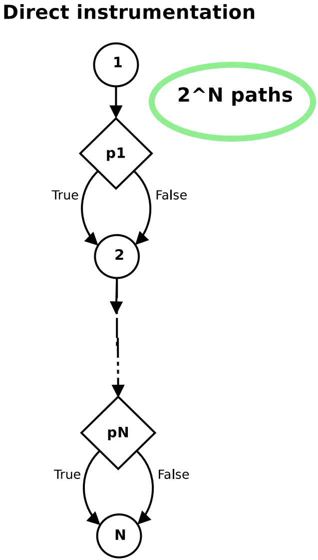
Direct instrumentation P' [APEX, Mutation DSE]



Covering label l ⇔ Covering branch `True`

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

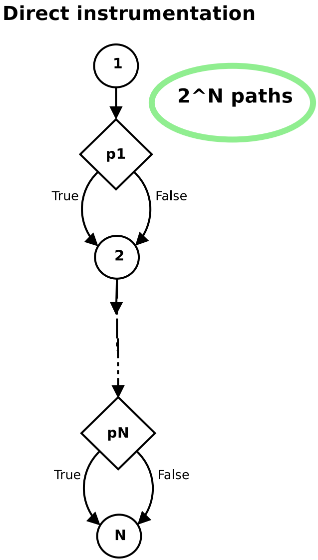
Direct instrumentation P' is not good enough



Direct instrumentation P' is not good enough

Non-tightness 1

- × P' has exponentially more paths than P



Direct instrumentation P' is not good enough

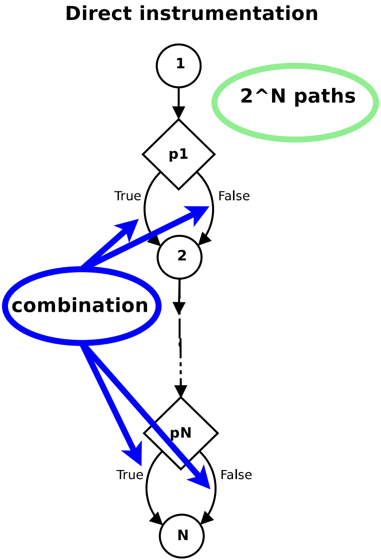
Non-tightness 1

× P' has exponentially more paths than P

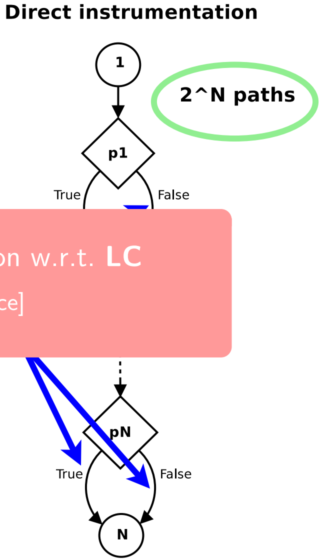
Non-tightness 2

× 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 P' is not good enough



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

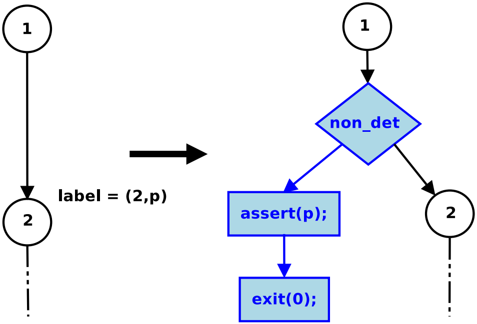
✗ dramatic overhead [theory & practice]

Our approach

The DSE* algorithm

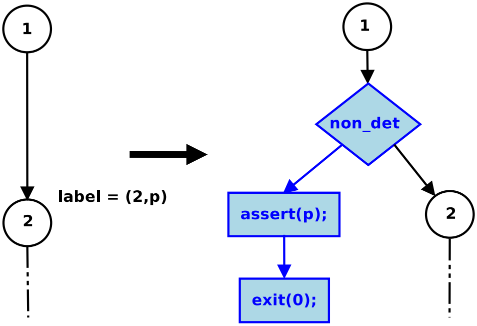
- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in a black-box manner

DSE* : Tight Instrumentation $P^\star$



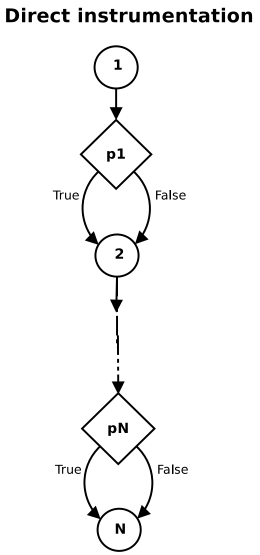
Covering label l $\Leftrightarrow$ Covering `exit(0)`

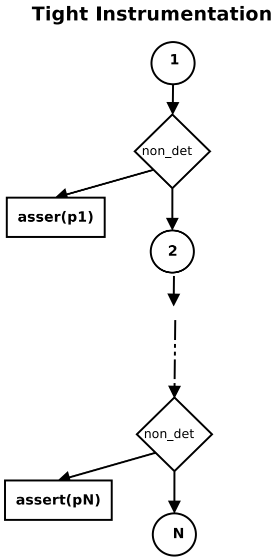
DSE* : Tight Instrumentation $P^\star$



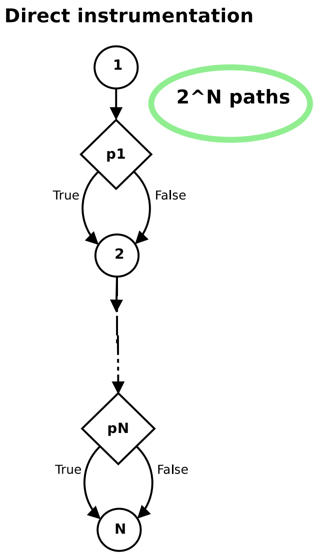
Covering label l $\Leftrightarrow$ Covering `exit(0)`

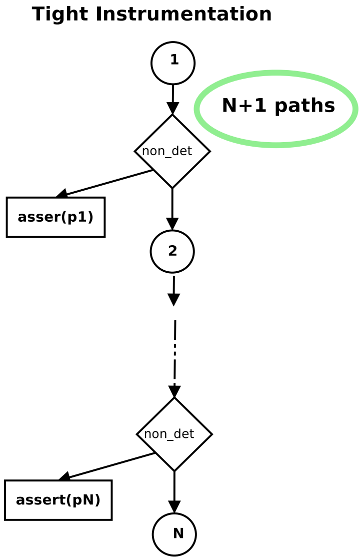
✓ sound & complete instrumentation w.r.t. **LC**


Direct instrumentation
1
p1
True
False
2
pN
True
False
N

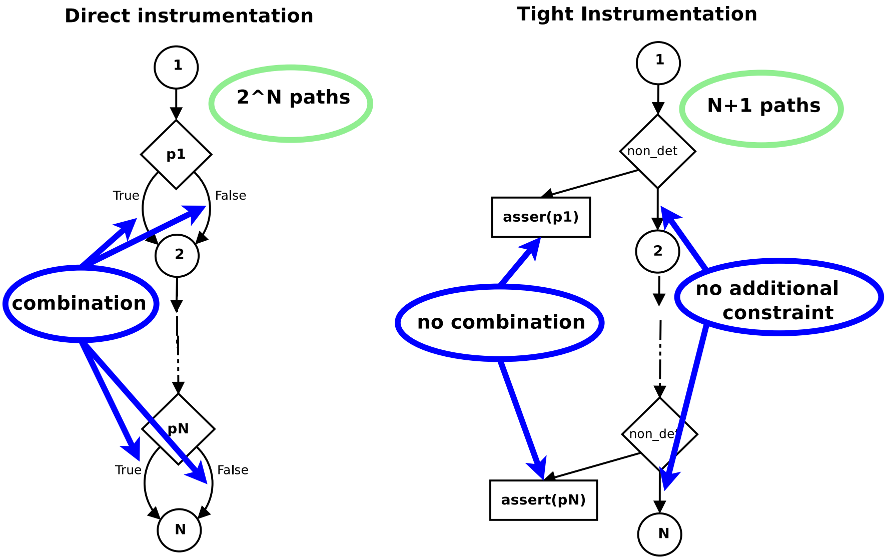

Tight Instrumentation
1
non_det
asser(p1)
2
non_det
assert(pN)
N

DSE* : Direct vs tight instrumentation, P' vs $P^\star$

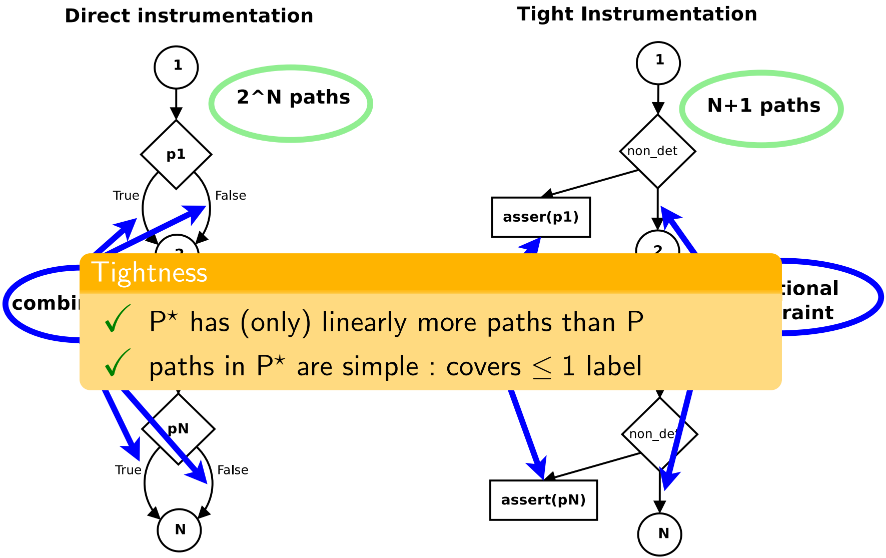




DSE* : Direct vs tight instrumentation, P' vs P^*



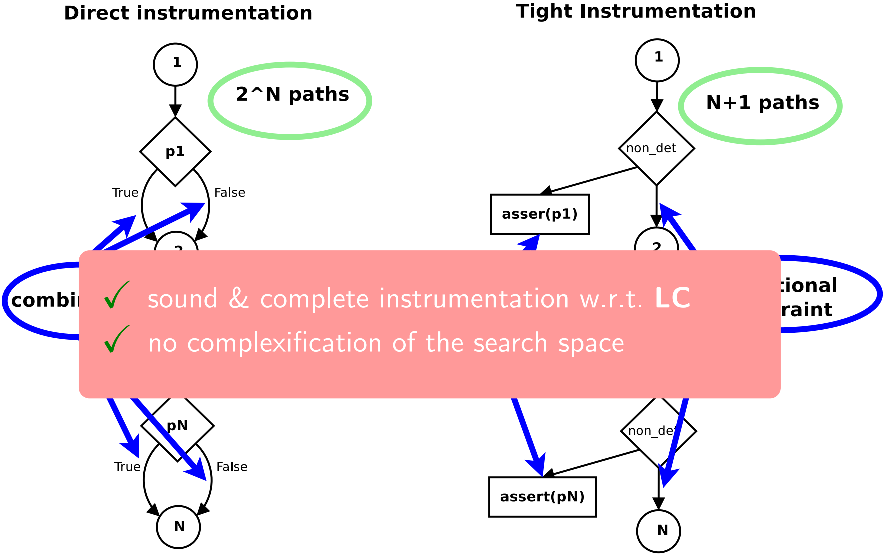
DSE* : Direct vs tight instrumentation, P' vs $P^\star$



Tightness

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

DSE* : Direct vs tight instrumentation, P' vs $P^\star$



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

DSE* : Iterative Label Deletion

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**

DSE* : Iterative Label Deletion

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 $P^\star$
- dynamic part : a slight modification of P'

DSE* : Iterative Label Deletion

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

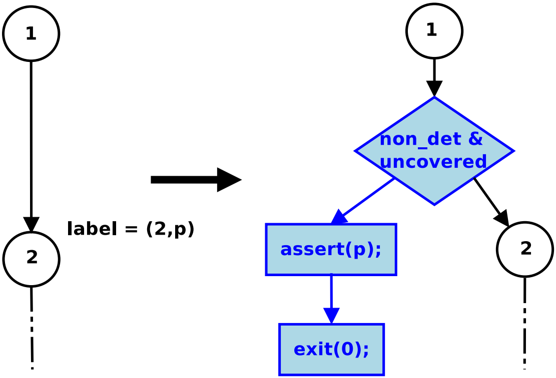
- keep a *covered/uncovered* status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates the status [truly requires DSE]

Implementation

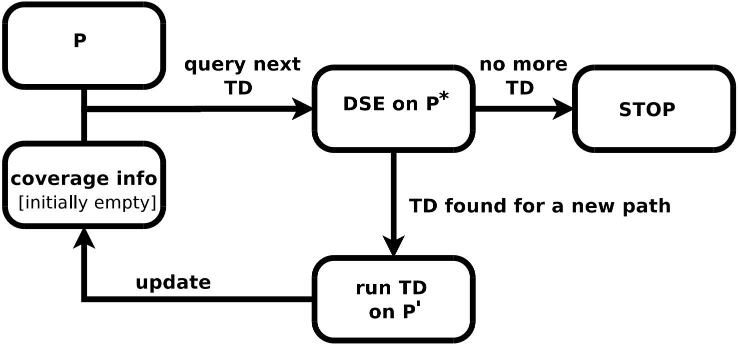
- symbolic part : a slight modification of $P^\star$
- dynamic part : a slight modification of P'

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

DSE* : Iterative Label Deletion (2)



DSE* : Iterative Label Deletion (3)



Summary

The DSE* algorithm

- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in black-box

Summary

The DSE* algorithm

tion"

Goals

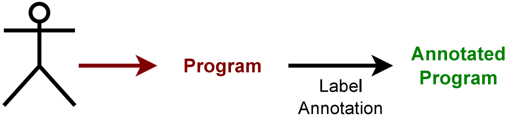
- ✓ GOAL1 : generic specification mechanism for coverage criteria
- ✓ GOAL2 : efficient integration into DSE

Outline

The LTEST toolset for labels [TAP 14]

Implementation 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



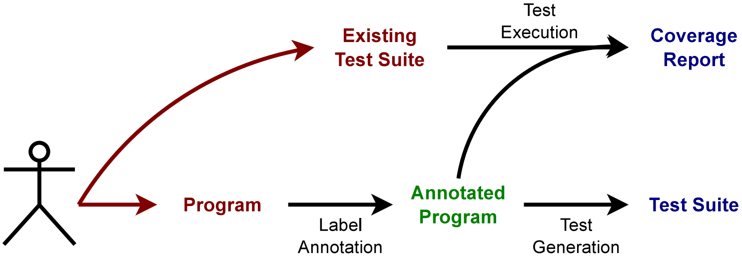
Supported criteria

- **DC**, **CC**, **MCC**
- **FC**, **IDC**, **WM**

Encoded with labels [ICST 2014]

- treated in a unified way
- easy to add new criteria

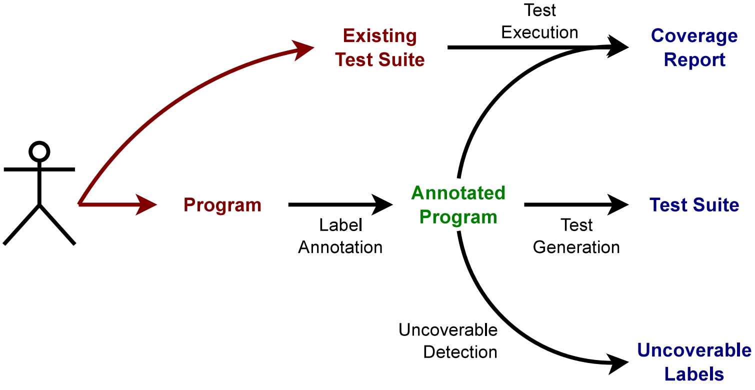
The LTest toolset for labels [TAP 14]



DSE* procedure [ICST 2014]

- DSE with native support for labels
- extension of PathCrawler

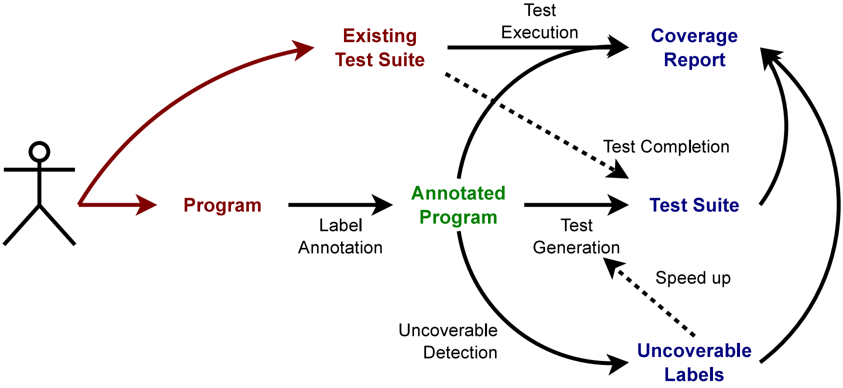
The LTEST toolset for labels [TAP 14]



Uses static analyzers from FRAMA-C

- sound detection of uncoverable labels

The LTEST toolset for labels [TAP 14]



Uses static analyzers from FRAMA-C

- sound detection of uncoverable labels

Service cooperation

- share label statuses
- `Covered`, `Infeasible`, ?

Experiments

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**

Experiments (2)

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%]

Experiments (2)

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%]

Experiments (2)

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

Outline

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

Disjunction

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

Safety

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

Hyperproperties

Hyperlabel Specification Language (HTOL)

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

l ::=	$\ell \triangleright B$	atomic label with bindings
B ::=	$\{v_1 \leftarrow\!\!\dashv e_1; \ldots\}$	bindings
h ::=	l	label
|	$[l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i} \}^\star\ l_n]$	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

LABEL

$$\frac{t \in TS \qquad t \rightsquigarrow_P^k \langle loc, s\rangle \qquad s \models \varphi \qquad \mathcal{E} \supseteq [\![B]\!]_s}{t \rightsquigarrow_{\mathcal{E}}^k \langle loc, \varphi\rangle \triangleright B \qquad \langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P \langle loc, \varphi\rangle \triangleright B}$$

GUARD

$$\frac{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h \qquad \mathcal{E} \models \psi}{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P \langle h \mid \psi\rangle}$$

CONJUNCTION

$$\frac{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_1 \qquad \langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_2}{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_1 \cdot h_2}$$

DISJUNCTION LEFT

$$\frac{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_1}{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_1 + h_2}$$

DISJUNCTION RIGHT

$$\frac{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_2}{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P h_1 + h_2}$$

SEQUENCE

$$\frac{\begin{array}{c} t \in TS \qquad \forall i \in [1,n],\ t \rightsquigarrow_{\mathcal{E}}^{k_i} l_i \qquad \forall i \in [1, n-1],\ k_i < k_{i+1} \\ \forall i \in [1, n-1],\ \forall j \in]k_i, k_{i+1}[,\ (loc_j, s_j) = P(t)_j \wedge \phi_i(\mathcal{E}, loc_j, s_j) \end{array}}{\langle TS, \mathcal{E}\rangle \overset{H}{\rightsquigarrow}_P [l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i} \}^\star\ l_n]}$$

Naming convention: TS test suite; $\mathcal{E}$ hyperlabel environment; h, h_1, h_2 hyperlabels; ψ hyperlabel guard predicate; n positive integer; $l_1, \ldots, l_n$ atomic labels with bindings; t test datum; $k, k_1, \ldots, k_n$ 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; $\phi_1, \ldots, \phi_n$ predicates over sequences of labels; φ label predicate; B hyperlabel bindings.

HTOL : Examples

- 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

HTOL : Examples

- 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

HTOL : Examples

- 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

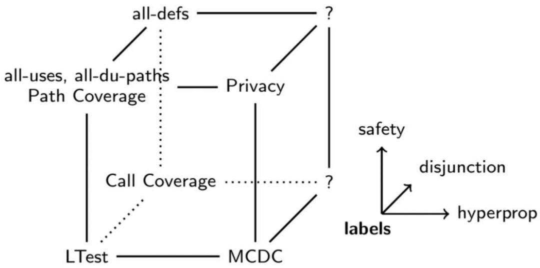
$$l \triangleq (loc_1, d) \triangleright \{c_1 \leftarrowtail \texttt{x==y};\ c_2 \leftarrowtail \texttt{a<b}\}$$

$$l' \triangleq (loc_1, \neg d) \triangleright \{c_1' \leftarrowtail \texttt{x==y};\ c_2' \leftarrowtail \texttt{a<b}\}$$

$$h_1 \triangleq \langle l \cdot l' \mid c_1 \neq c_1' \wedge c_2 = c_2' \rangle$$

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:**



HTOL : Expressiveness and support

- **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)

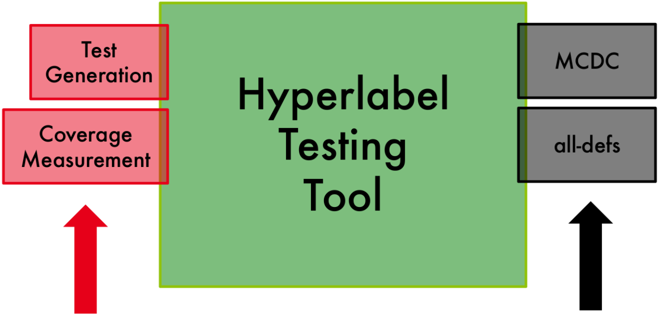
Tool name	BBC	FC	DC	CC	DCC	GACC	MCDC	MCC	BP	Other
Gcov	✓	✓	✓							0/19
Bullseye		✓			✓					0/19
Parasoft	✓	✓	✓	✓			✓		✓	0/19
Semantic Designs		✓	✓							0/19
Testwell CTC++	✓	✓			✓		✓			0/19
LTest	✓	✓	✓	✓	✓	✓		✓		4/19
Hyper-LTest	✓	✓	✓	✓	✓	✓	✓	✓	✓	18/19

- **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)

Impact of a generic toolset like LTest



Pluggable automation techniques for hyperlabel objectives

Pluggable hyperlabel generation modules for criteria

New criteria get all techniques + New techniques get all criteria
for free thanks to hyperlabels

Outline

Summary

Goal = express and support a large class of coverage criteria

Results

- Labels : a well-defined and expressive specification mechanism for coverage criteria
- DSE* : an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]
- Hyperlabels : an extension of labels, capable to express almost all existing coverage criteria

Summary

Goal = express and support a large class of coverage criteria

Results

- Labels : a well-defined and expressive specification mechanism for coverage criteria
- DSE* : an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]
- Hyperlabels : an extension of labels, capable to express almost all existing coverage criteria

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

✓ very powerful approach to (white box) test generation

✓ arguably one of the most wide-spread uses of formal methods in "common software"

Summary

Goal = express and support a large class of coverage criteria

Results

- Labels : a well-defined and expressive specification mechanism for coverage criteria
- DSE* : an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]
- Hyperlabels : an extension of labels, capable to express almost all existing coverage criteria

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

✓ very powerful approach to (white box) test generation

✓ arguably one of the most wide-spread uses of formal methods in "common software"

✗ support only basic coverage criteria

Summary

Goal = express and support a large class of coverage criteria

Results

- Labels : a well-defined and expressive specification mechanism for coverage criteria
- DSE* : an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]
- Hyperlabels : an extension of labels, capable to express almost all existing coverage criteria

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

✓ very powerful approach to (white box) test generation

✓ arguably one of the most wide-spread uses of formal methods in "common software"

✓ can be efficiently extended to a large class of coverage criteria

Future work

- 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 LTool in industry