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...

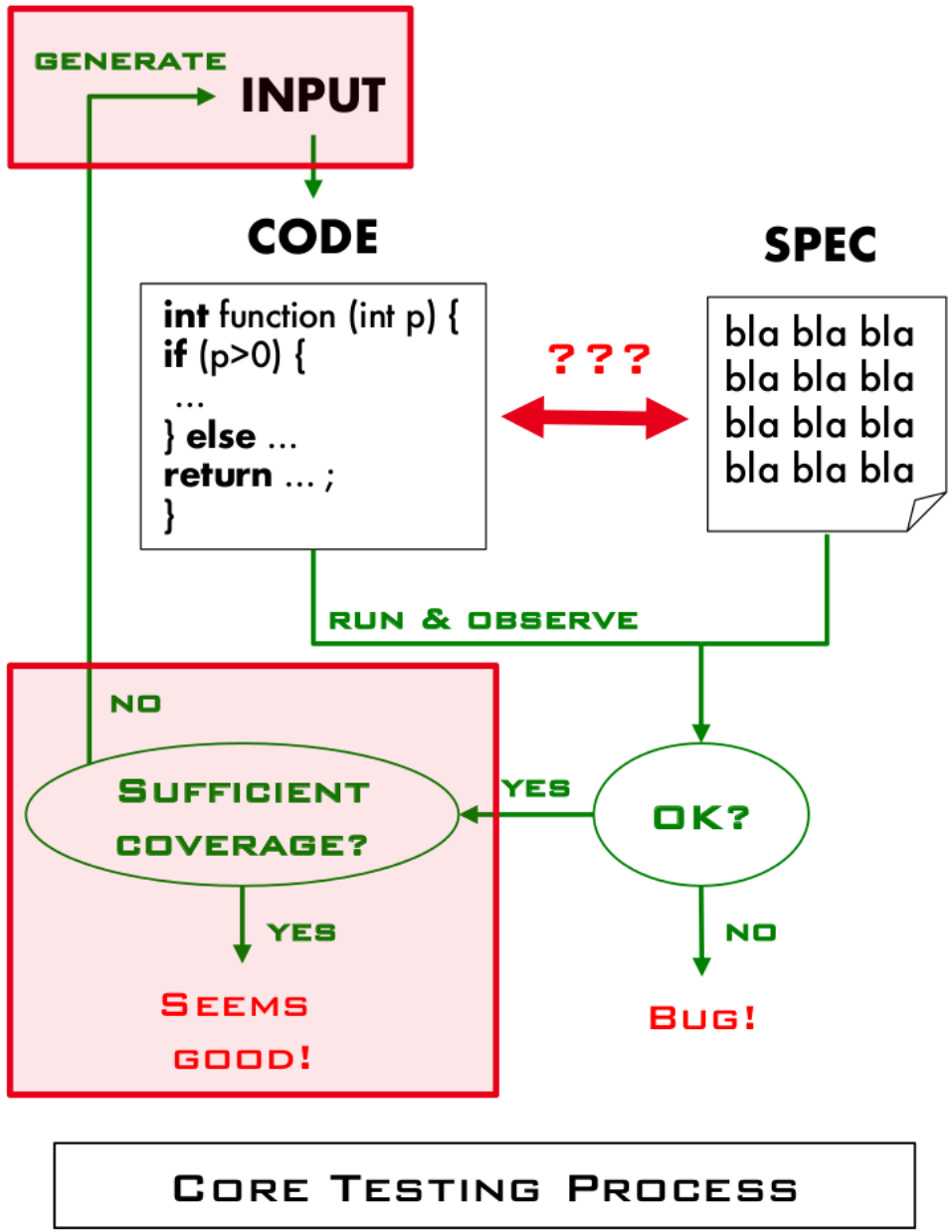
CEA List & Thales Research and Technology

LRI, Univ. Paris-Sud, February 7, 2020

Context: White-Box Testing

Testing process

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



Context: White-Box Testing

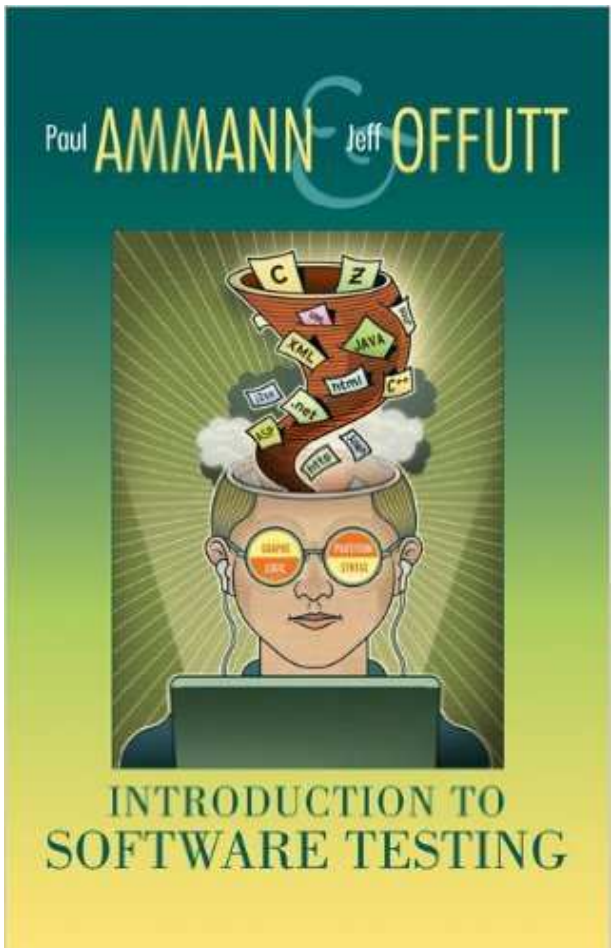
- Framework: white-box software testing process
- Automate test suite generation & coverage measure
- Coverage criterion = objectives to be fulfilled by the test suite
- Criterion guides automation
- Can be part of industrial normative requirements

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



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, ISOLA 2018]
[Marcozzi et al., ICST 2017 (res.), ICST 2017 (tool), 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

8]

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

8]

Outline

Labels and the notion of simulation (1/2)

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;
```

Labels and the notion of simulation (2/2)

Basic definitions

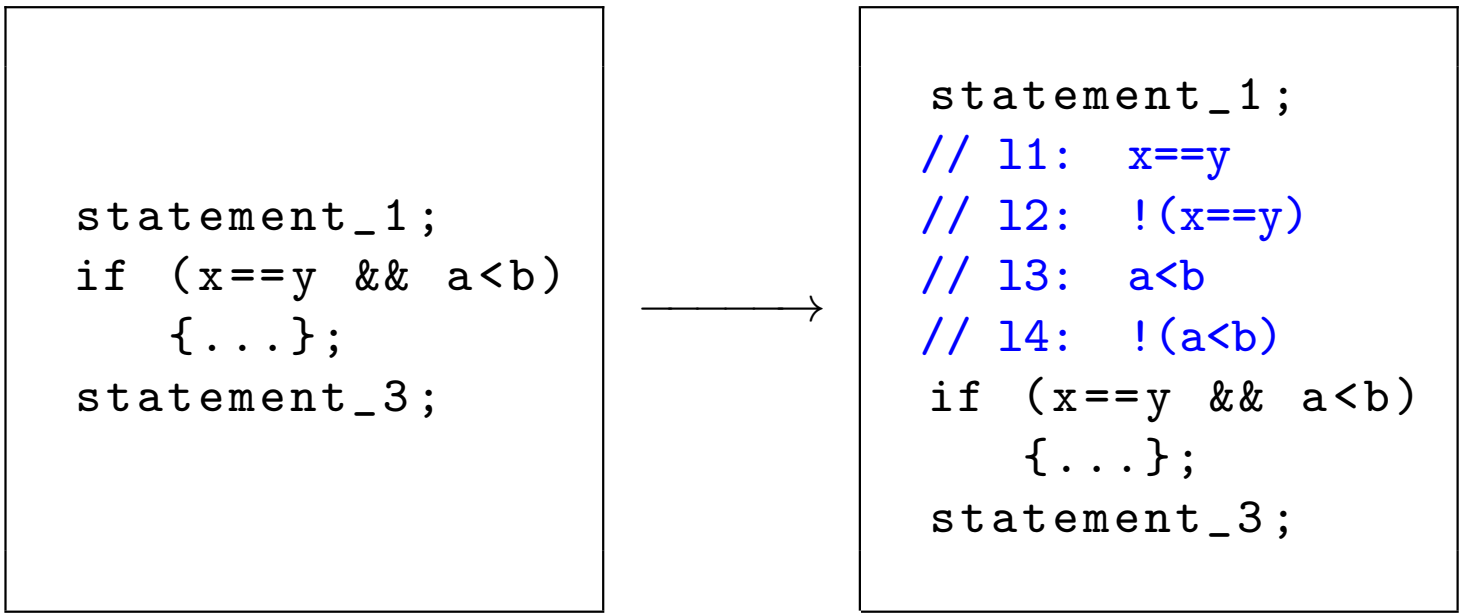
- a test datum t covers l if $P(t)$ reaches loc and satisfies φ
- new criterion **LC** label coverage: requires to cover the labels

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**.

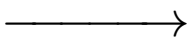
Simulation of coverage criteria by labels: CC



Condition Coverage (**CC**)

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: MCC

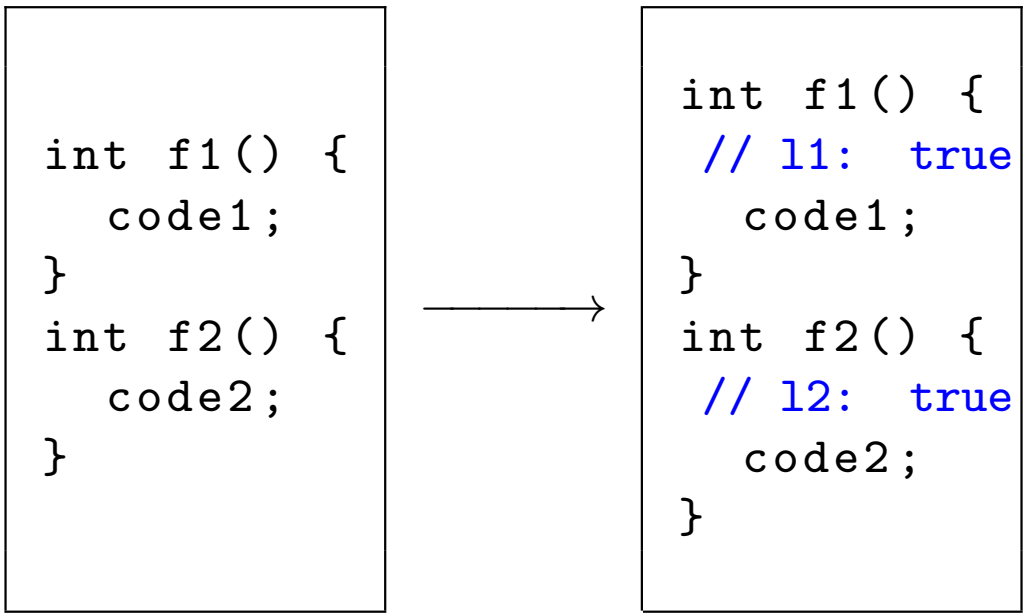
```
statement_1;
if (x==y && a<b)
   {...};
statement_3;
```



```
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**)

Simulation of coverage criteria by labels: FC



Function Coverage (**FC**)

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, weak mutations $\mathbf{WM}_O$ *can be simulated by* **LC**.

Measuring the coverage of a test suite

- 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 to obtain P'
  - 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: $\leq |TS| \cdot |M| \cdot \max_{m \in M, t \in TS}(m(t))$

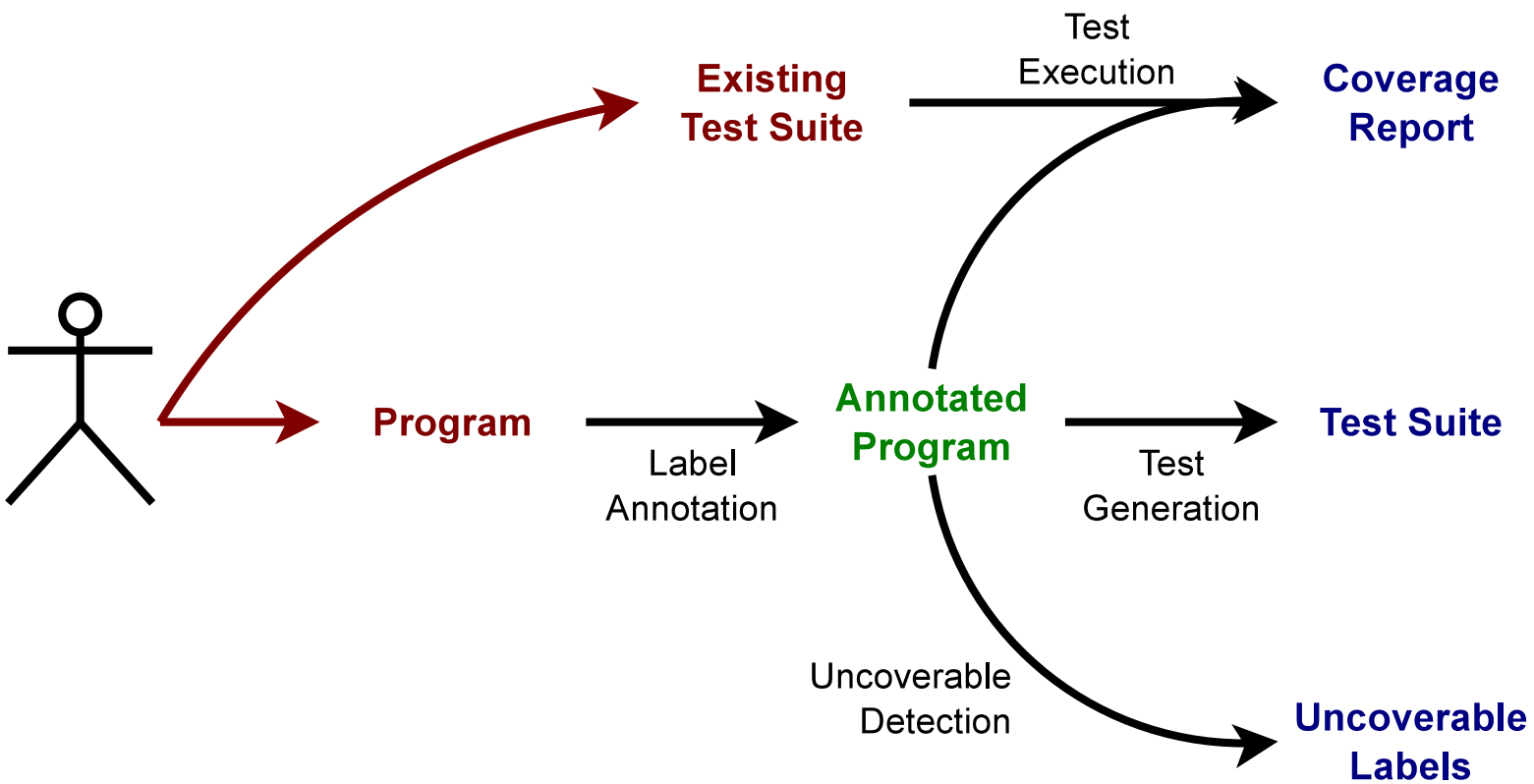
Outline

The LTest toolset for labels

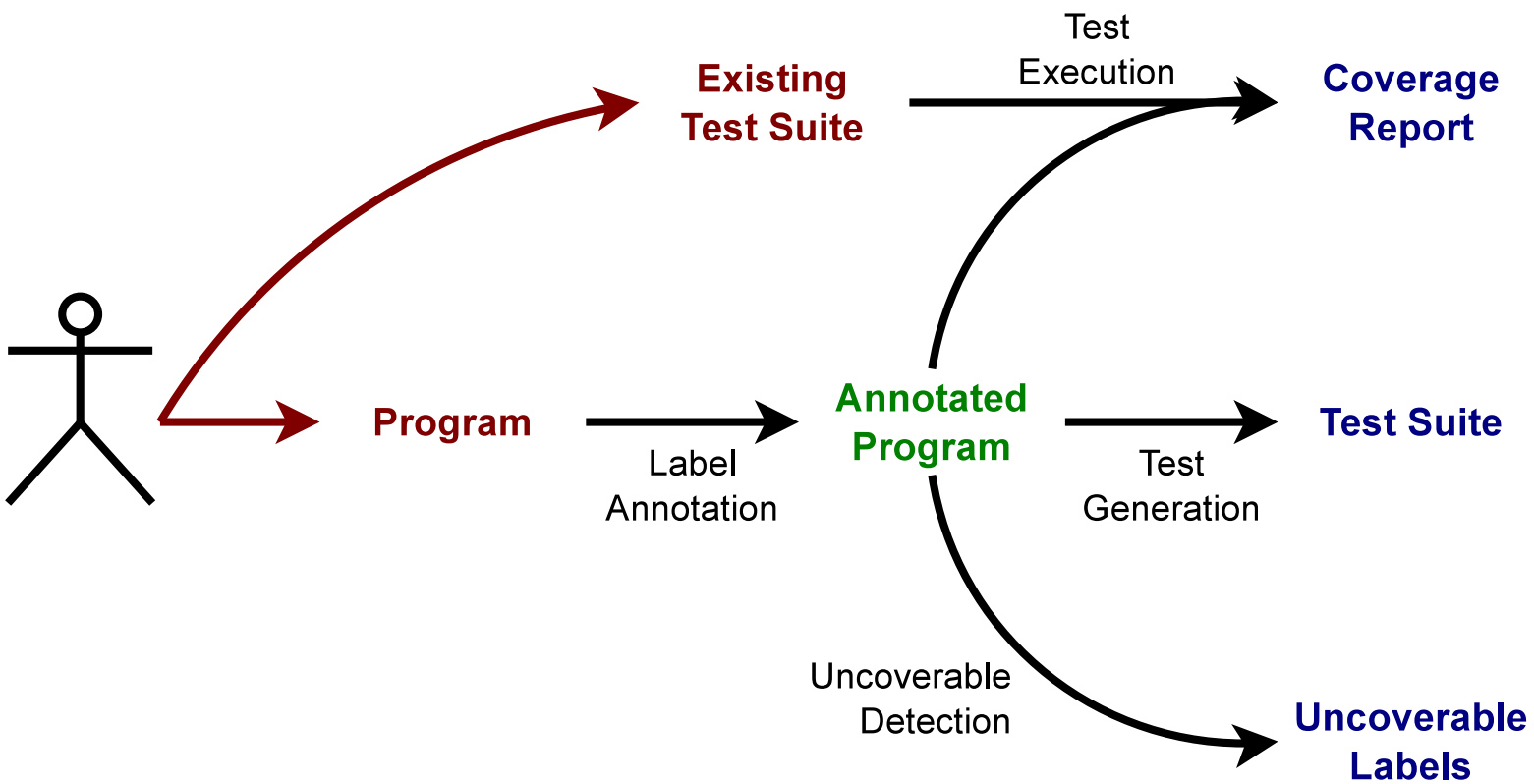
LTest is implemented on top of Frama-C

- 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

Outline

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

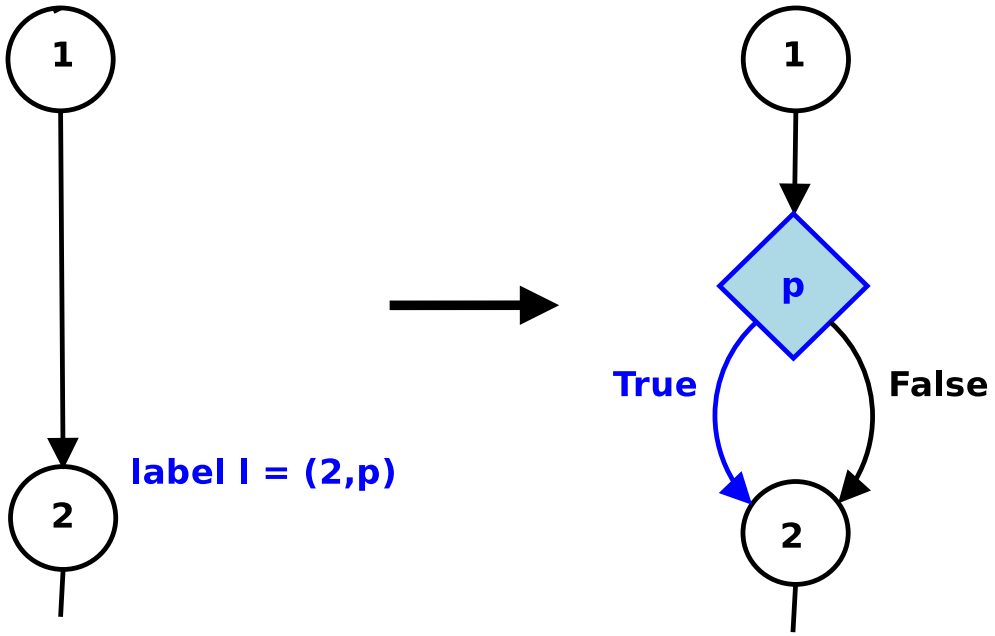
Dynamic Symbolic Execution

- ✓ 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

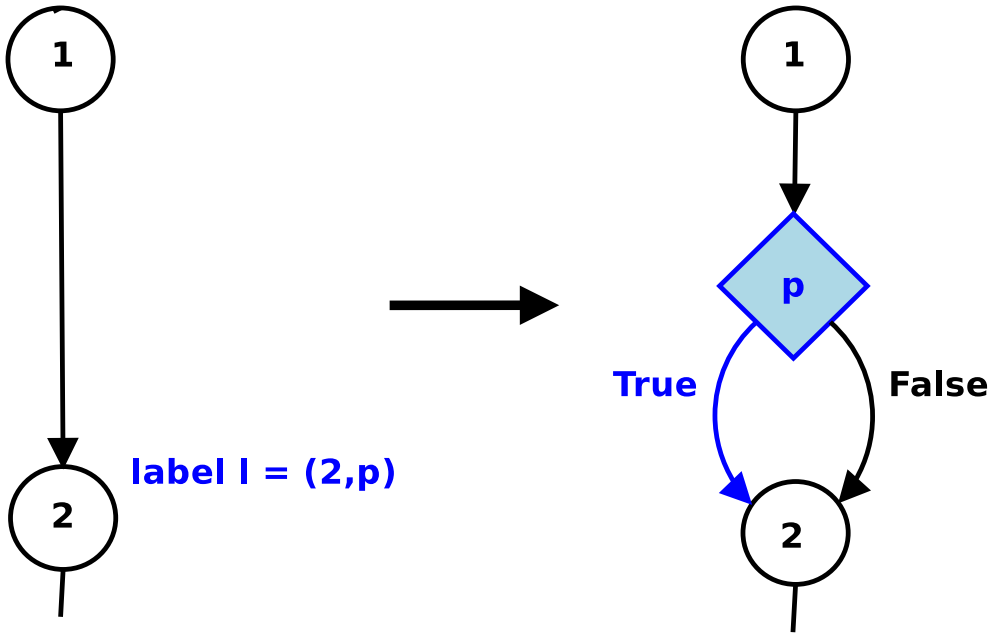
- well-known problem
- recent efforts in this direction through instrumentation
  [Active Testing, Mutation DSE, Augmented DSE]
- limitations:
  - exponential explosion of the search space [APEX: 272x avg]
  - very implementation-centric mechanisms
  - unclear expressiveness

Direct instrumentation P' [APEX, Mutation DSE]



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

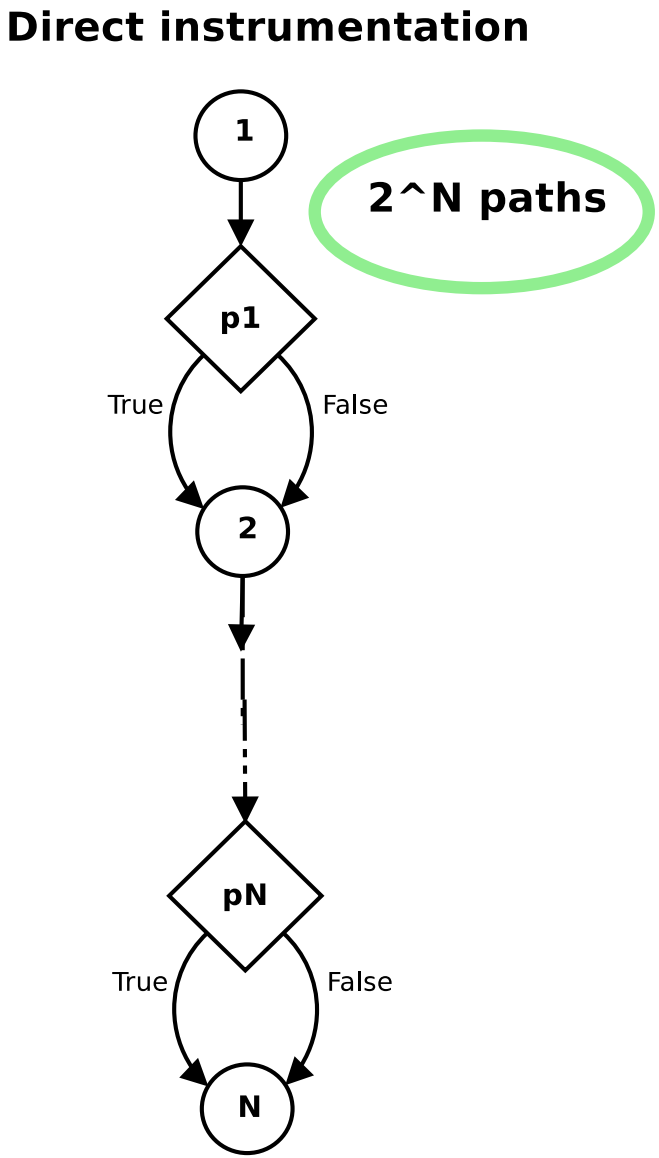
Direct instrumentation P' [APEX, Mutation DSE]



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

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

Direct instrumentation P' is not good enough

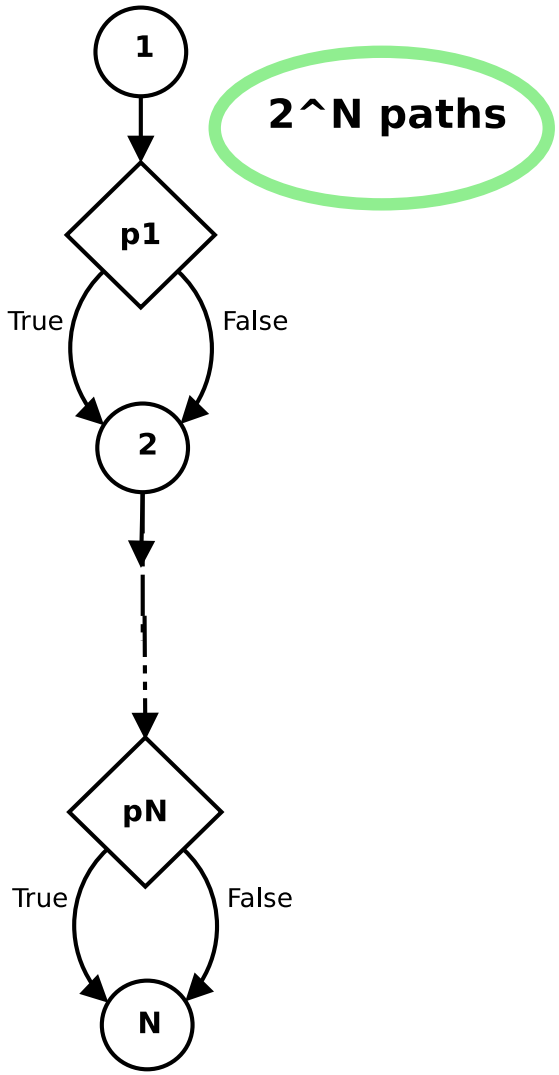


Direct instrumentation P' is not good enough

Non-tightness 1

- ✗ P' has exponentially more paths than P

Direct instrumentation



Direct instrumentation P' is not good enough

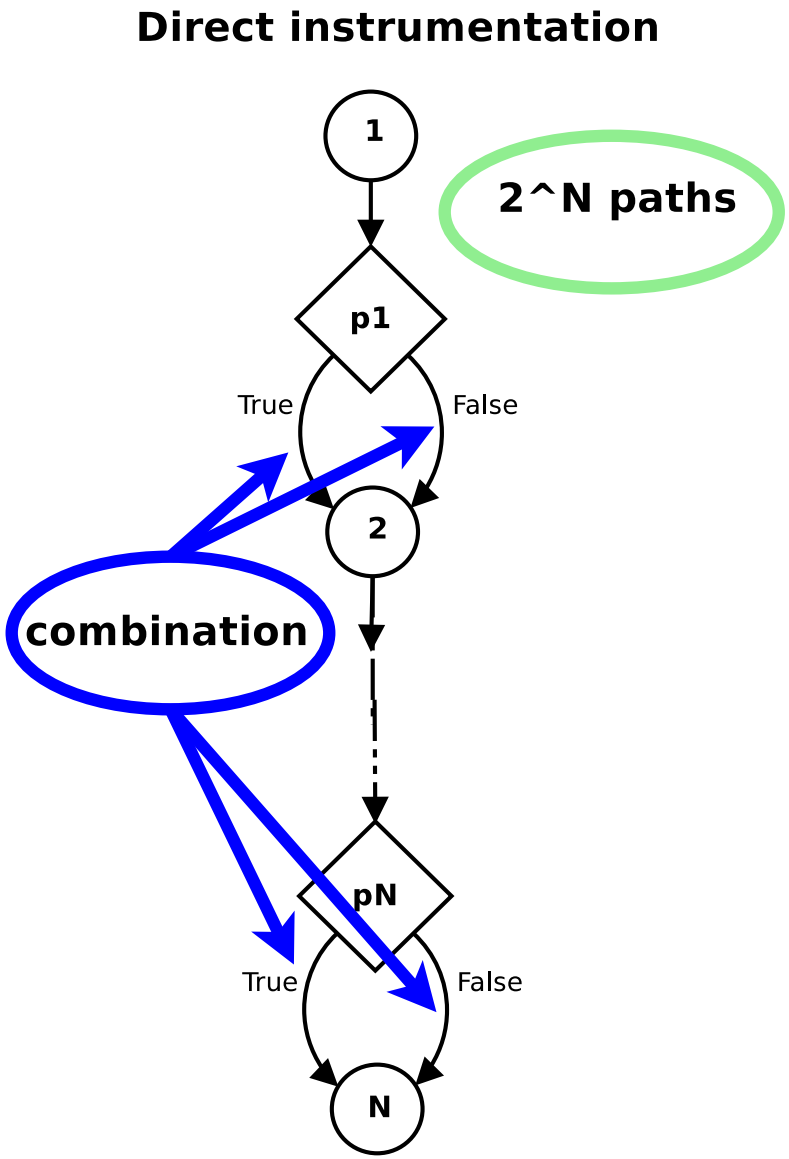
Non-tightness 1

× P' has exponentially more paths than P

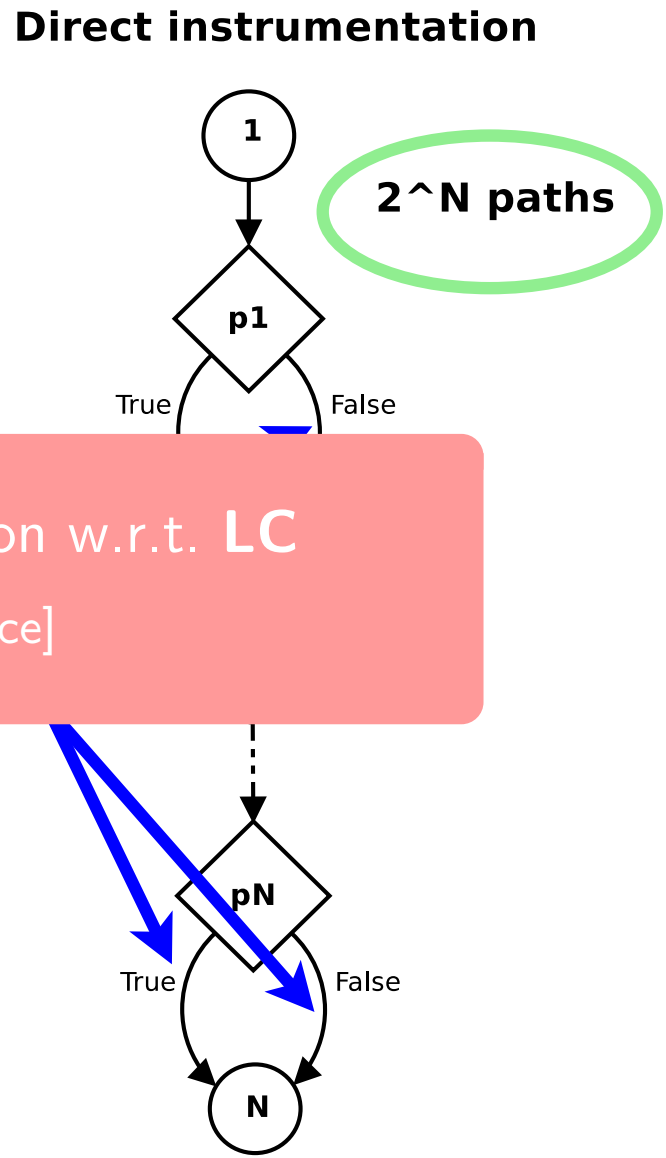
Non-tightness 2

× 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**

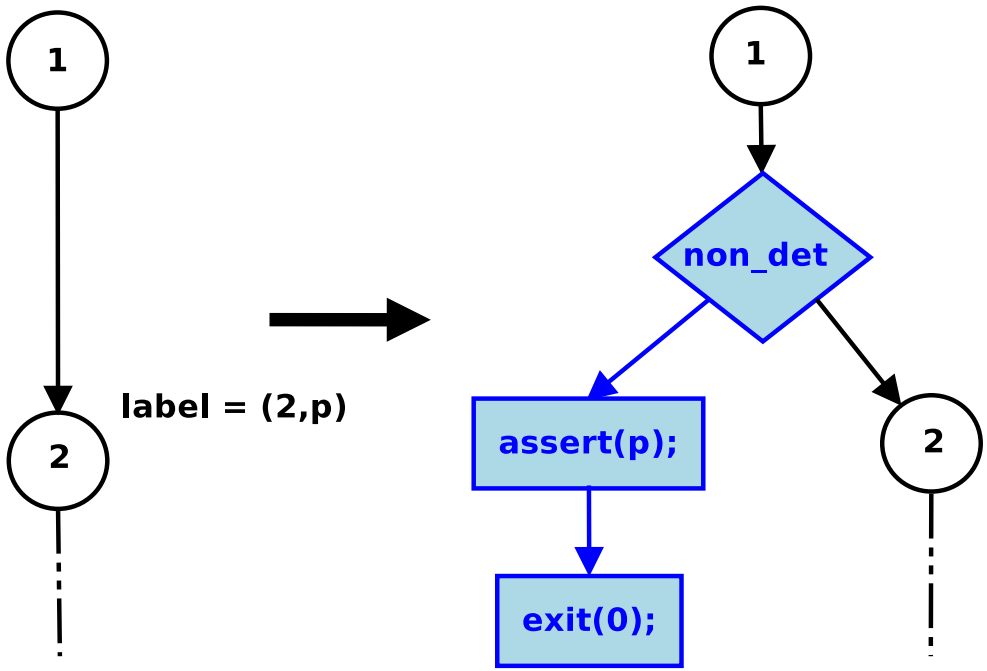
✗ dramatic overhead [theory & practice]

Our approach

The DSE* algorithm

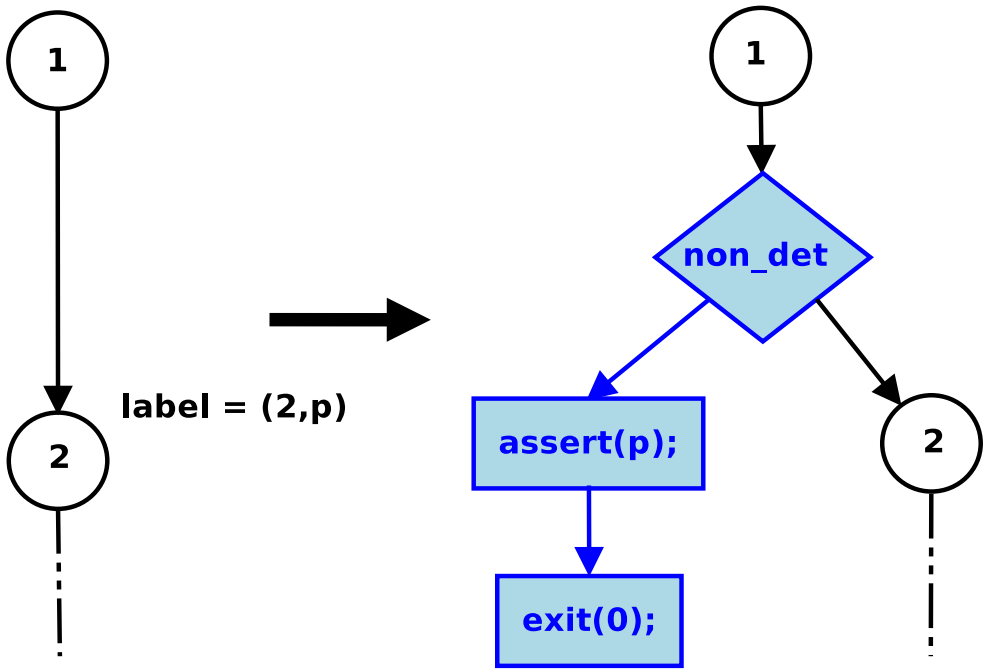
- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion: discards some redundant paths
- Both techniques can be implemented in a black-box manner

DSE*: Tight Instrumentation $P^\star$



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

DSE*: Tight Instrumentation $P^\star$

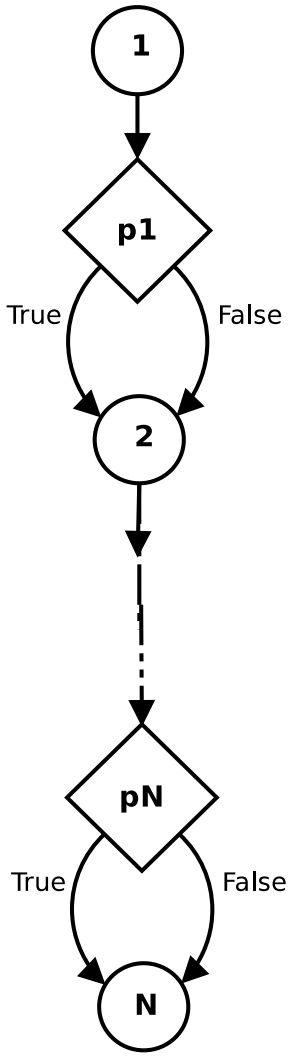


Covering label l $\Leftrightarrow$ Covering `exit(0)`

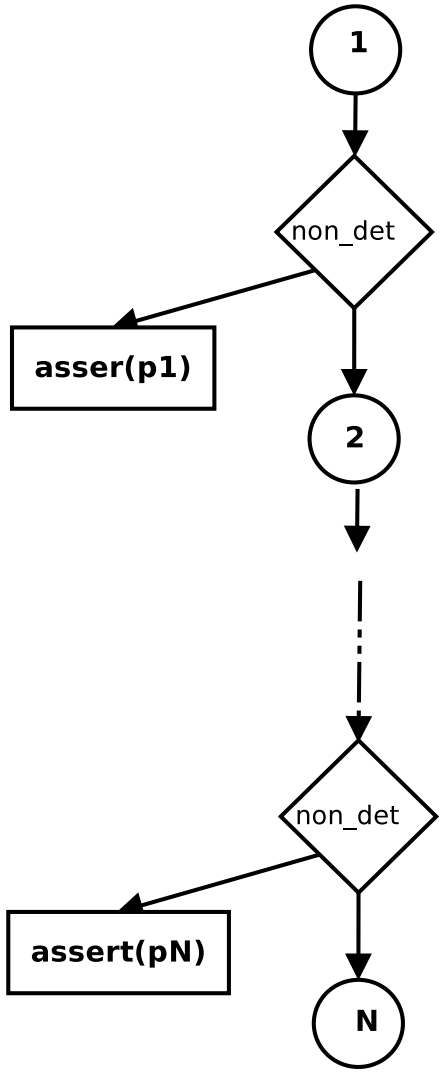
✓ sound & complete instrumentation w.r.t. **LC**

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

Direct instrumentation

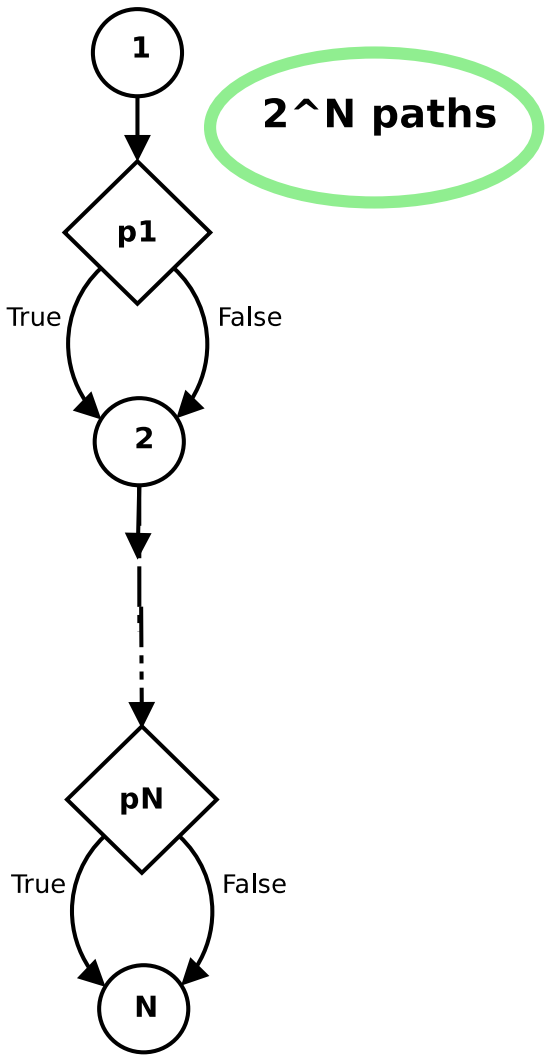


Tight Instrumentation

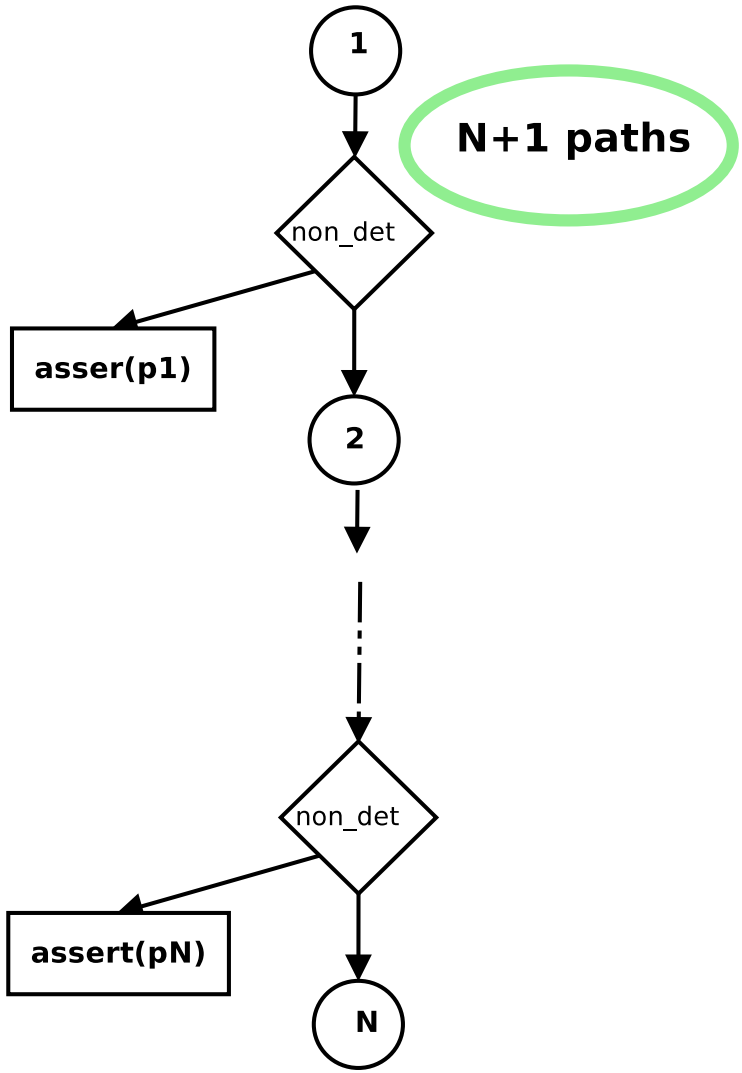


DSE*: Direct vs tight instrumentation, P' vs $P^\star$

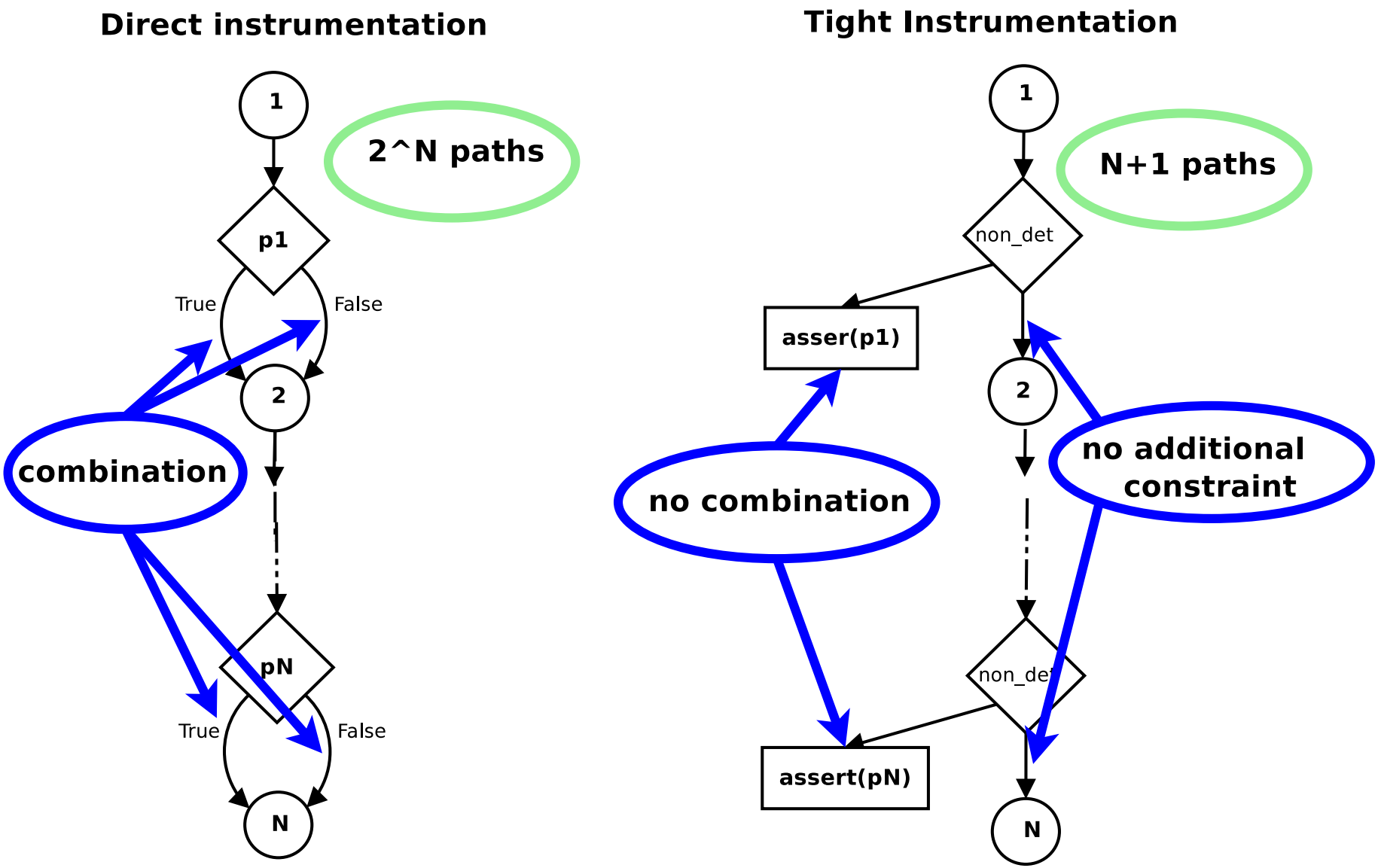
Direct instrumentation



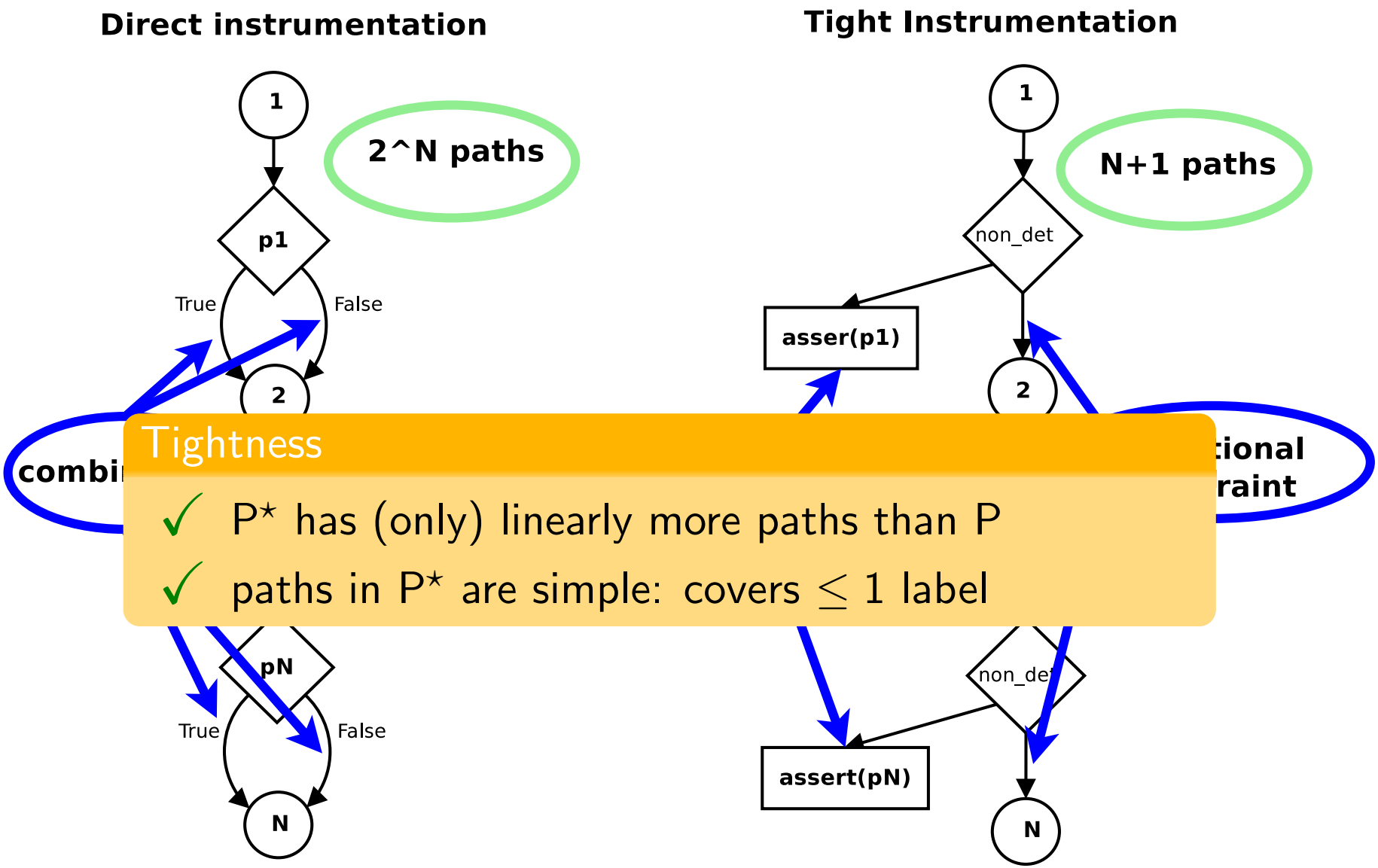
Tight Instrumentation



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



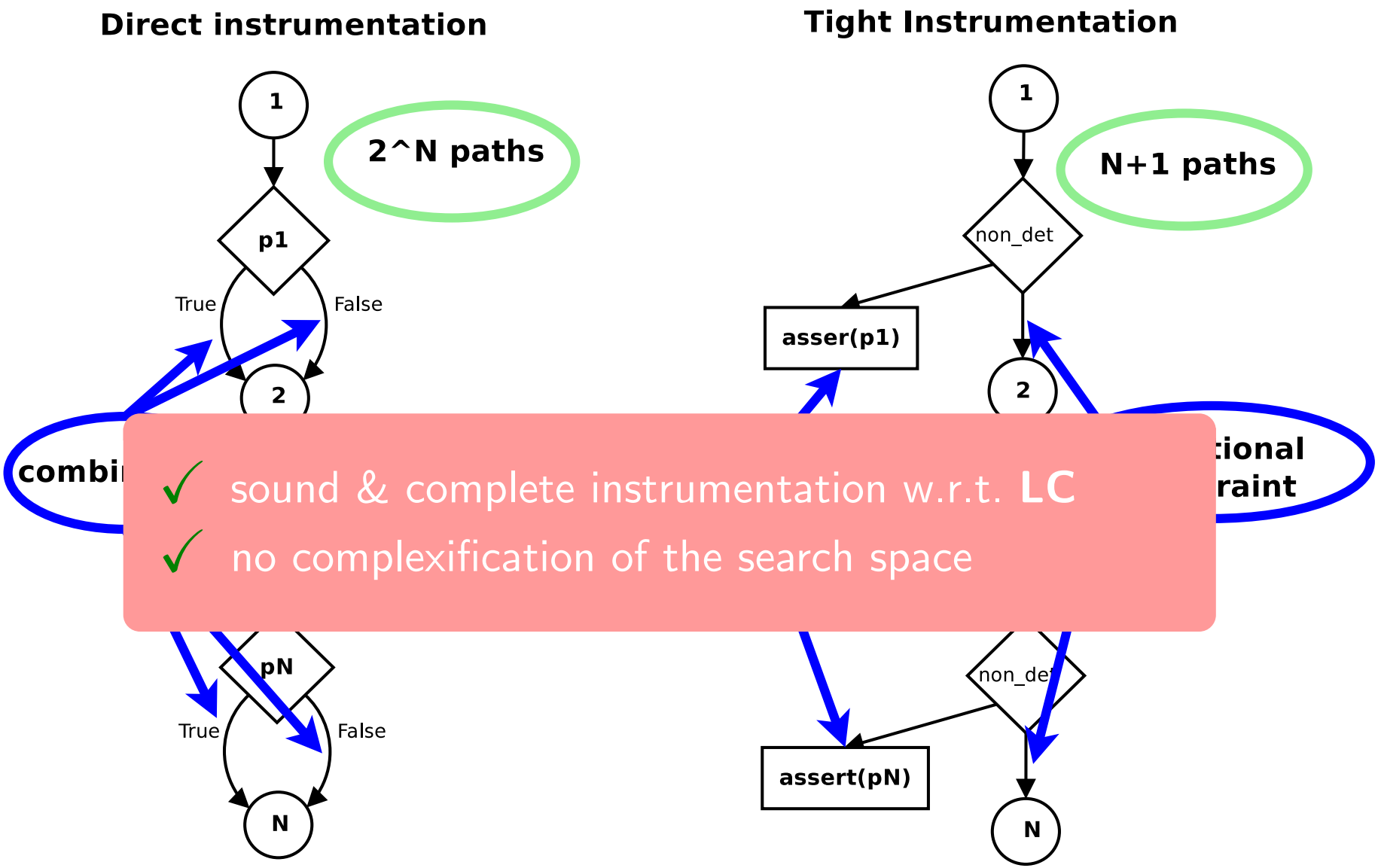
DSE*: Direct vs tight instrumentation, P' vs $P^\star$



Tightness

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

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



- ✓ 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 $P^\star$ 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

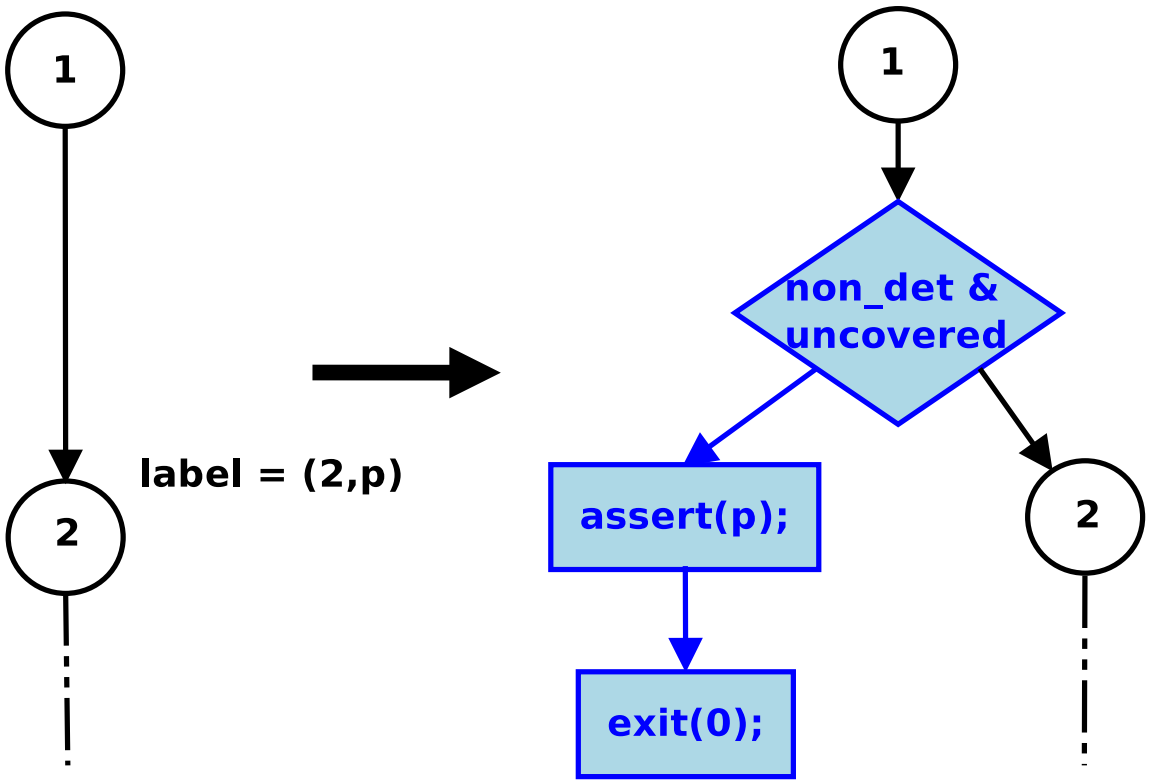
- keep a *covered/uncovered* status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates the status [truly requires DSE]

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)



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

DSE* dramatically improves test generation performances

- APEX reports an average overhead >272x [Jamrozik et al, TAP 2013]
- DSE* leads to an average overhead of 2.4x [Bardin et al, ICST 2014]

PathCrawler/LTest: Towards an industrial adoption

MERCE, a research branch of Mitsubishi Electric, developed additional modules for automatic annotation of labels, generation of stubs, generation of test sheets targeting their industrial needs

MERCE evaluated the complete automatic tool

- on industrial code of 80,000 lines, 1,300 functions in 150 files
- the tool was able to parse and annotate 100% of the files and generate test cases for 86% of functions
- the generation took $<$ 1 day instead of $\sim$230 days manually

Those very good results are very encouraging
for pushing the technology in business units
[Bardin et al, ISOLA 2018]

Outline

Uncoverable test objectives in testing

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)

Detection goals

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

Focus: checking assertion validity

- **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;
}
```

LUncov Methodology: Combine VA $\oplus$ WP

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⊕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⊕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;
}
```

LUncov: Results and Experiments

- 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 ($\leq$ 1s/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)

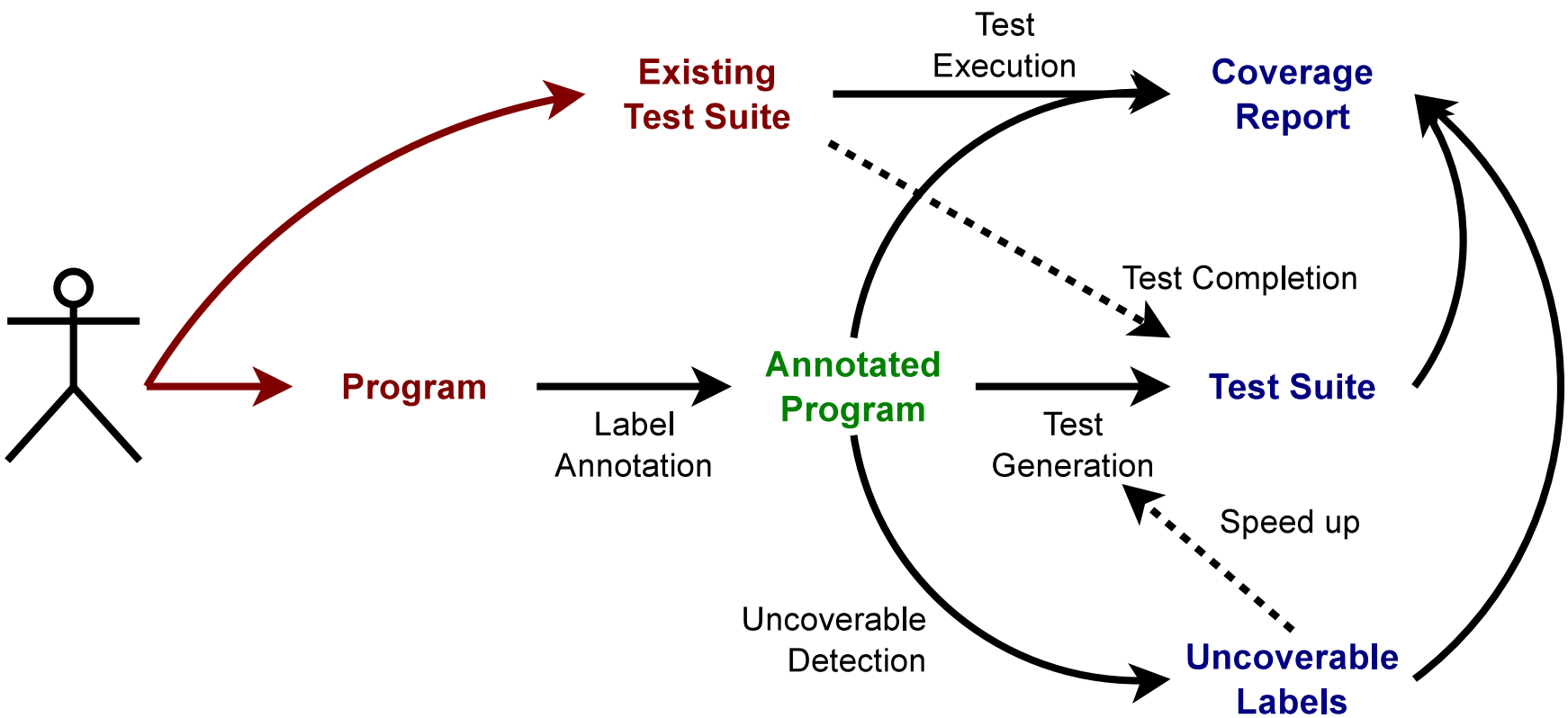
[Bardin et al. ICST 2014, TAP 2014, ICST 2015]

Detecting polluting objectives

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`, `?`

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

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

Sequences

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	::=	$\{v_1 \leftarrow e_1; \ldots\}$	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

Hyperlabel Specification Language (HTOL) – Semantics

Formal Semantics:

$$\textsc{Label}\quad \frac{t \in TS \qquad t \rightsquigarrow_P^k \langle loc, s\rangle \qquad s \vDash \varphi \qquad \mathcal{E} \supseteq [\![B]\!]_s}{t \rightsquigarrow_{\mathcal{E}}^k \langle loc, \varphi\rangle \rhd B \qquad \langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P \langle loc, \varphi\rangle \rhd B}$$

$$\textsc{Guard}\quad \frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h \qquad \mathcal{E} \vDash \psi}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P \langle h \mid \psi\rangle}$$

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

$$\textsc{Disjunction Left}\quad \frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 + h_2}$$

$$\textsc{Disjunction Right}\quad \frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_2}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 + h_2}$$

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

Naming convention: TS test suite; $\mathcal{E}$ hyperlabel environment; h, h_1, h_2 hyperlabels; ψ hyperlabel guard predicate; n positive integer; $l_1, \ldots, l_n$ 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

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

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

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

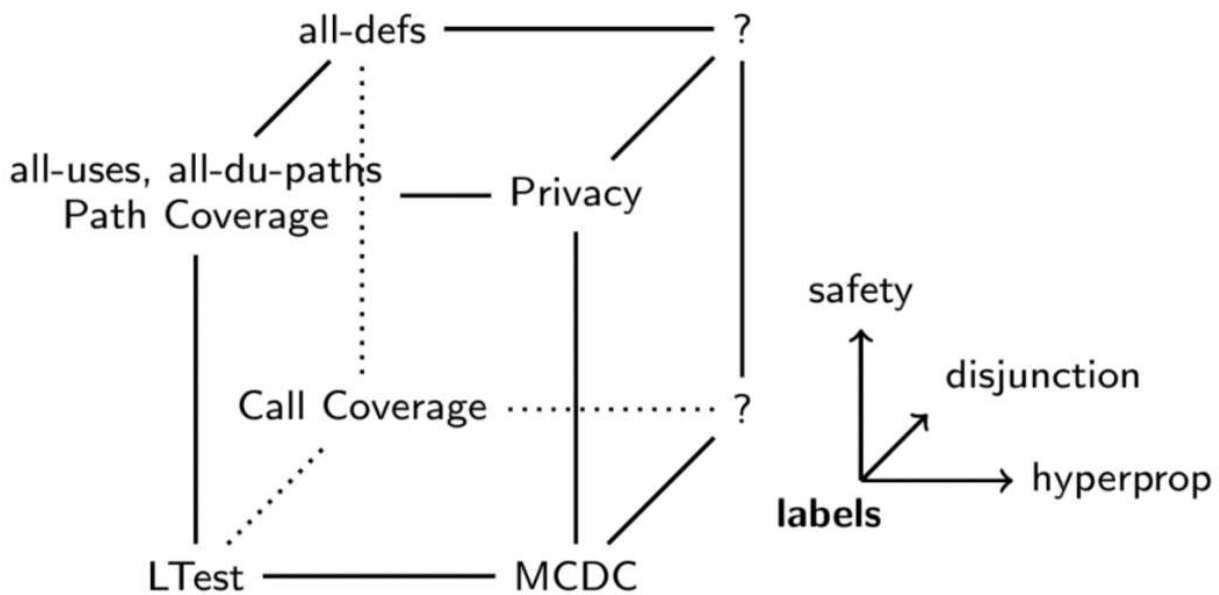
$$l \triangleq (loc_1, d) \triangleright \{c_1 \hookleftarrow \texttt{x==y}; c_2 \hookleftarrow \texttt{a<b}\}$$

$$l' \triangleq (loc_1, \neg d) \triangleright \{c'_1 \hookleftarrow \texttt{x==y}; c'_2 \hookleftarrow \texttt{a<b}\}$$

$$h_1 \triangleq \langle l \cdot l' \mid c_1 \neq c'_1 \wedge c_2 = c'_2 \rangle$$

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)

Outline

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 [✓]

Future work

- An efficient dedicated support of hyperlabels in test generation (DSE)
- Further optimizations of LTest (e.g. detection of uncoverable hyperlabels)
- Combinations with fuzzing, genetic algorithms and machine learning
- Developing the emerging interest for LTest in industry