Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria

Sébastien Bardin, Nikolai Kosmatov and François Cheynier

CEA LIST, Software Safety Lab
(Paris-Saclay, France)

Dynamic Symbolic Execution

Dynamic Symbolic Execution [dart, cute, 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

Dynamic Symbolic Execution [dart, cute, 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.

  $$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P(v)} \text{ follows } \sigma$$
- can be used for bounded-path testing !
- old idea, recent renew interest [requires powerful solvers]

Dynamic Symbolic Execution

Dynamic Symbolic Execution [dart, cute, 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.

  $$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P(v)} \text{ follows } \sigma$$
- can be used for bounded-path testing !
- old idea, recent renew interest [requires powerful solvers]

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

- interleave dynamic and symbolic executions
- drive the search towards feasible paths for free
- give hints for relevant under-approximations [robustness]

Dynamic Symbolic Execution (2)

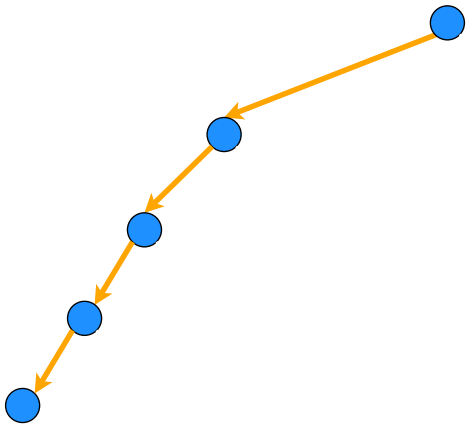
input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover

Dynamic Symbolic Execution (2)

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

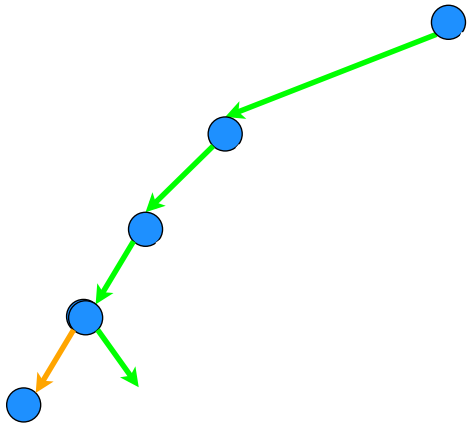
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Dynamic Symbolic Execution (2)

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

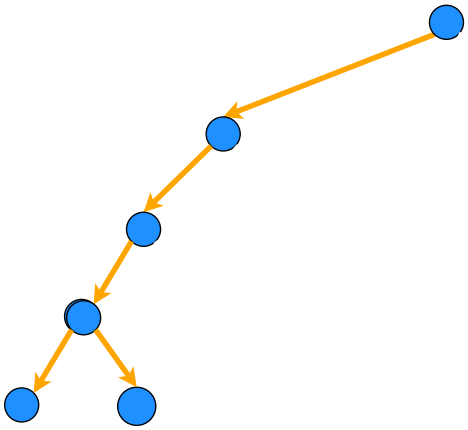
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Dynamic Symbolic Execution (2)

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

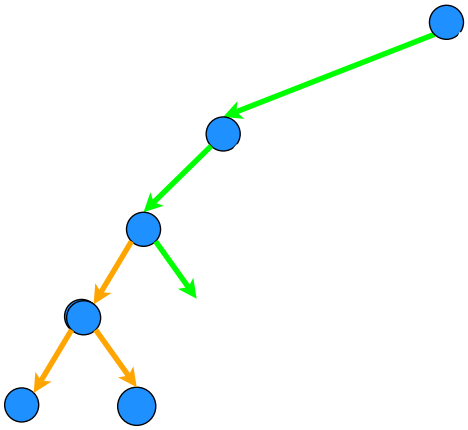
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Dynamic Symbolic Execution (2)

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

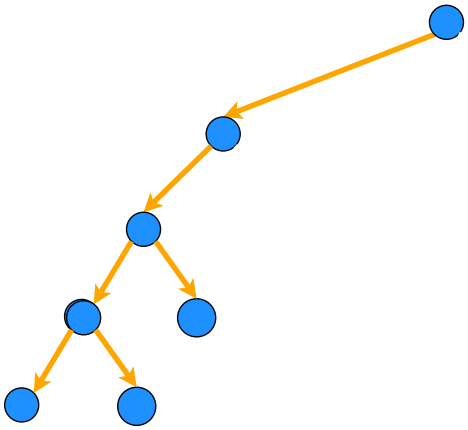
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Dynamic Symbolic Execution (2)

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

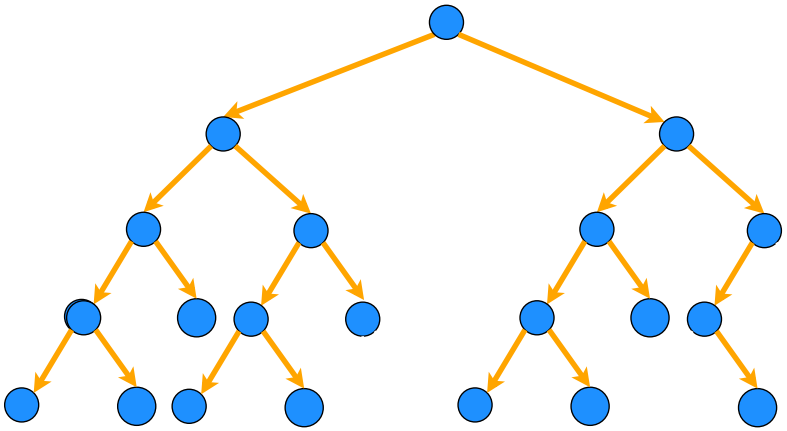
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Dynamic Symbolic Execution (2)

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

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path 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"
- ✗ support only basic coverage criteria (**IC**, **DC**)

DSE is limited in the following cases :

- generate *TS* achieving a given coverage criterion
- generate a "good" *TS* for an external oracle
  [functional correctness, security, performance, etc.]

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"
- ✗ support only basic coverage criteria (**IC**, **DC**)

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

Our results

Labels : a well-defined 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 blackbox

Implem. in PATHCRAWLER

- huge savings compared to existing approaches
- handle labels for only a low overhead (2x average, up to 7x)

Outline

- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion

Labels

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

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

Basic definitions

- an annotated program is a pair $\langle P, L \rangle$, with L set of labels
- a test datum t covers l if $P(t)$ reaches loc and satisfies φ
- new criterion **LC** (label coverage) for annotated programs

Notations

- $t \rightsquigarrow_{\langle P,L \rangle} l$ for "t covers l"
- $TS \rightsquigarrow_{\langle P,L \rangle}$ **LC** for "TS covers all labels of $\langle P, L \rangle$"

Criteria simulation

Goal = reasoning about the relative expressiveness of **LC**

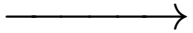
A labelling function ψ maps a program P into an annotated program $\psi(P) \triangleq \langle P, L \rangle$

Definition (Simulation)

A coverage criterion **C** *can be simulated by* **LC** if there exists a labelling function ψ such that for any program P and any test suite TS, we have : $TS \rightsquigarrow_P \mathbf{C}$ iff $TS \rightsquigarrow_{\psi(P)} \mathbf{LC}$.

Simulation of standard coverage criteria (1)

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

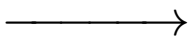


```
statement_1;
// l1: x==y && a<b
// l2: !(x==y && a<b)
if (x==y && a<b)
    {...};
statement_3;
```

Decision Coverage (**DC**)

Simulation of standard coverage criteria (2)

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

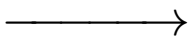


```
statement_1;
// l1: x==y
// l2: !(x==y)
// l3: a<b
// l4: !(a<b)
if (x==y && a<b)
    {...};
statement_3;
```

Condition Coverage (**CC**)

Simulation of standard coverage criteria (3)

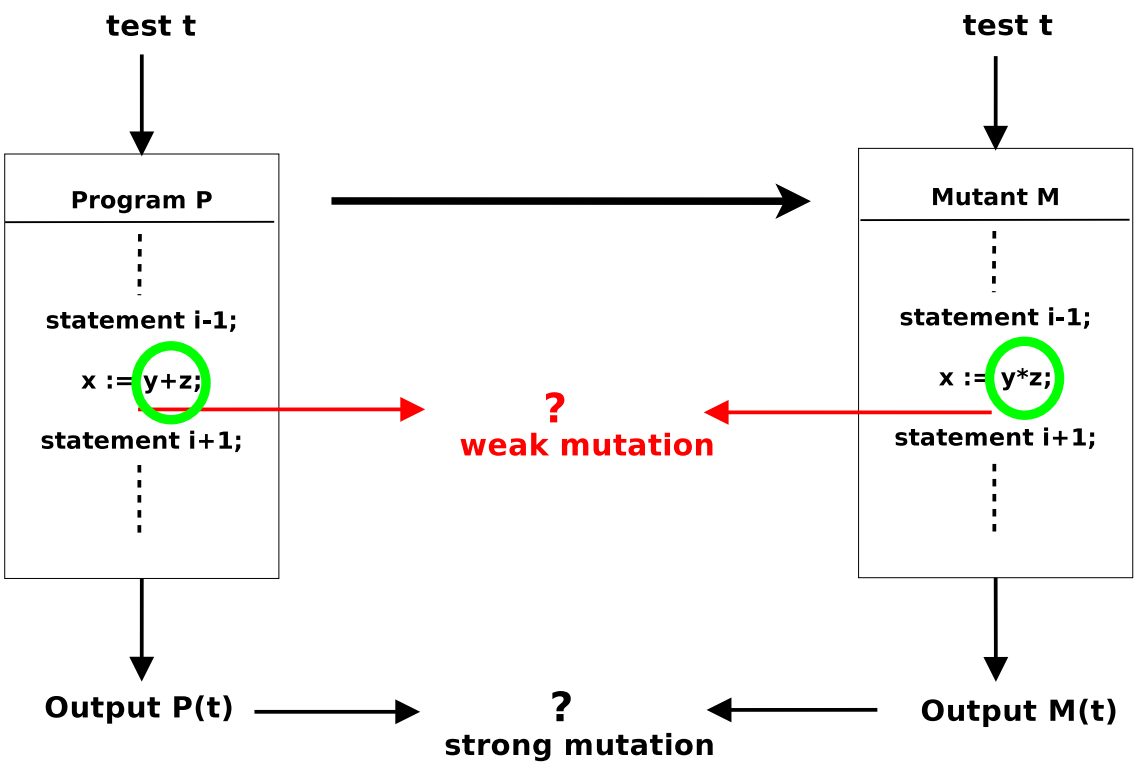
```
statement_1;
if (x==y && a<b)
    {...};
statement_3;
```



```
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 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

From weak mutants to labels

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

From weak mutants to labels

One label per mutant

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`

Theorem

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

Mutation inside a decision

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

Beware : no side-effect inside labels

From weak mutants to labels

One label per mutant

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`

Theorem

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

Mutation inside a decision

- `if (cond)`
  - add label

Beware : no side-e

Theorem

More simulations :
IC, **DC**, **FC**, **CC**, **MCC**,
Input Domain Partition,
Run-Time Error

From weak mutants to labels

One label per mutant

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`

Theorem

For any
operator

Mutation in

- `if (cond)`
  - add label

Beware : no side-e

Theorem

More simulations :
IC, **DC**, **FC**, **CC**, **MCC**,
Input Domain Partition,
Run-Time Error

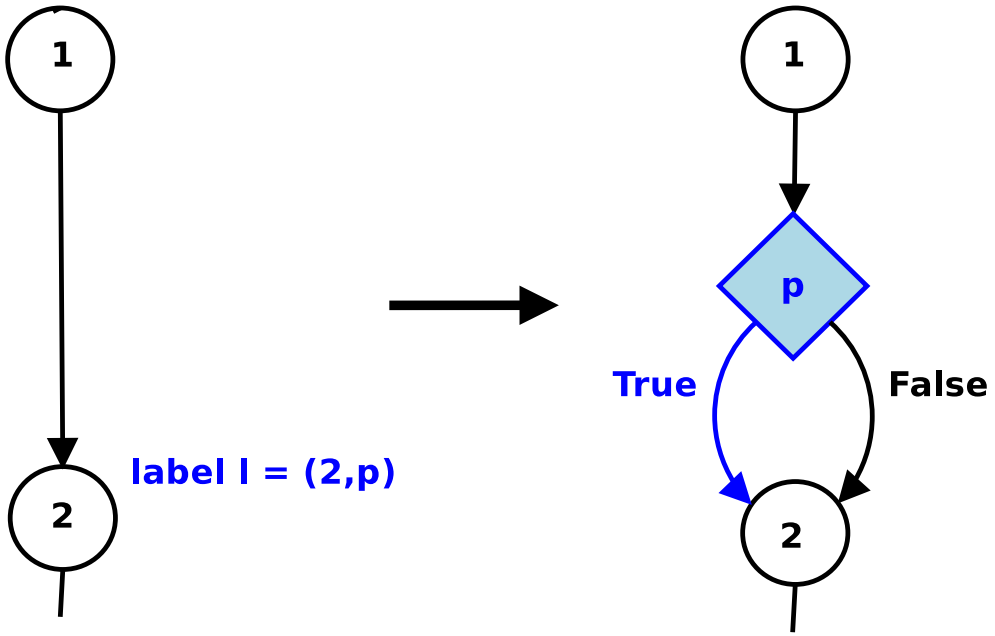
Goals

- ✓ OBJ1 : generic specification mechanism for coverage criteria
- OBJ2 : efficient integration into DSE

Outline

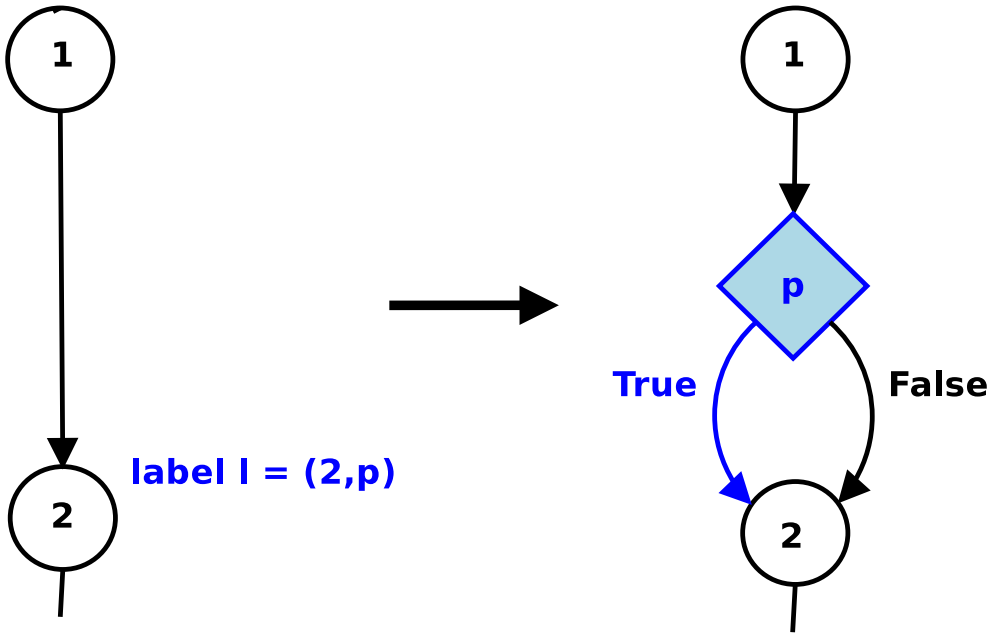
- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion

Direct instrumentation [APEX, Mutation DSE]



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

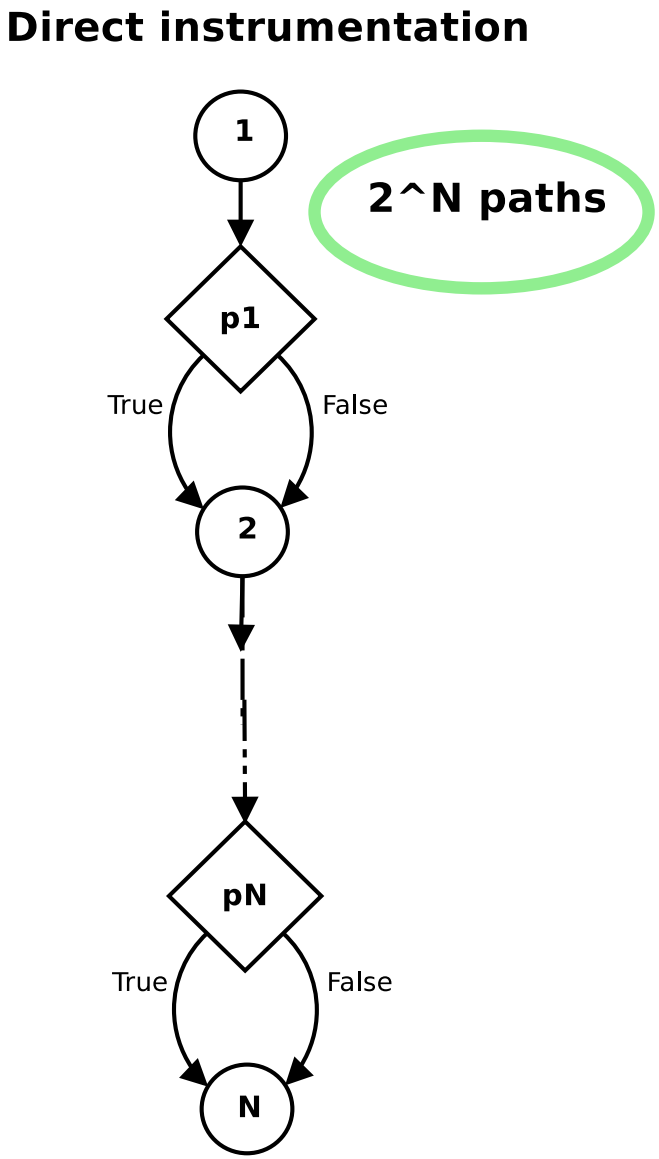
Direct instrumentation [APEX, Mutation DSE]



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

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

Direct instrumentation is not good enough

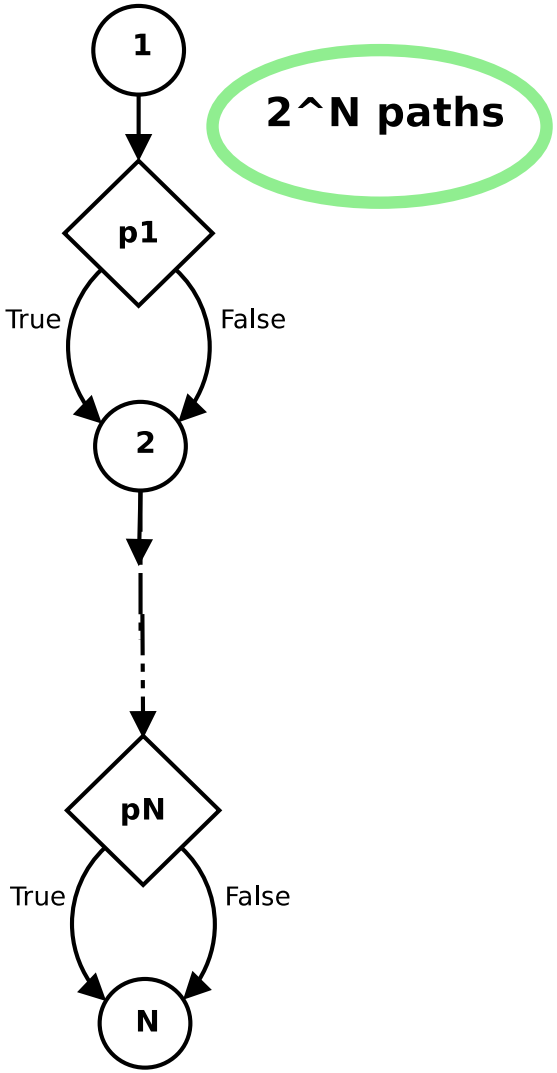


Direct instrumentation is not good enough

Non-tightness 1

× P' has exponentially more paths than P

Direct instrumentation



Direct instrumentation is not good enough

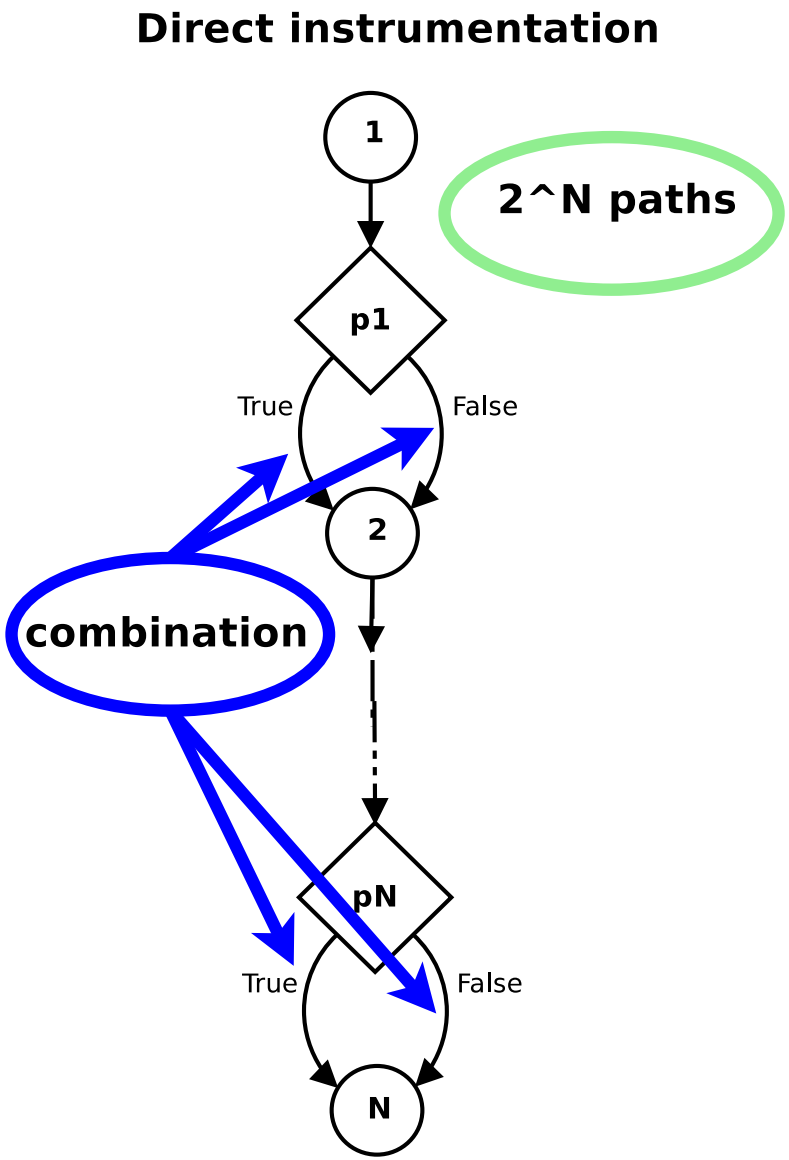
Non-tightness 1

× P' has exponentially more paths than P

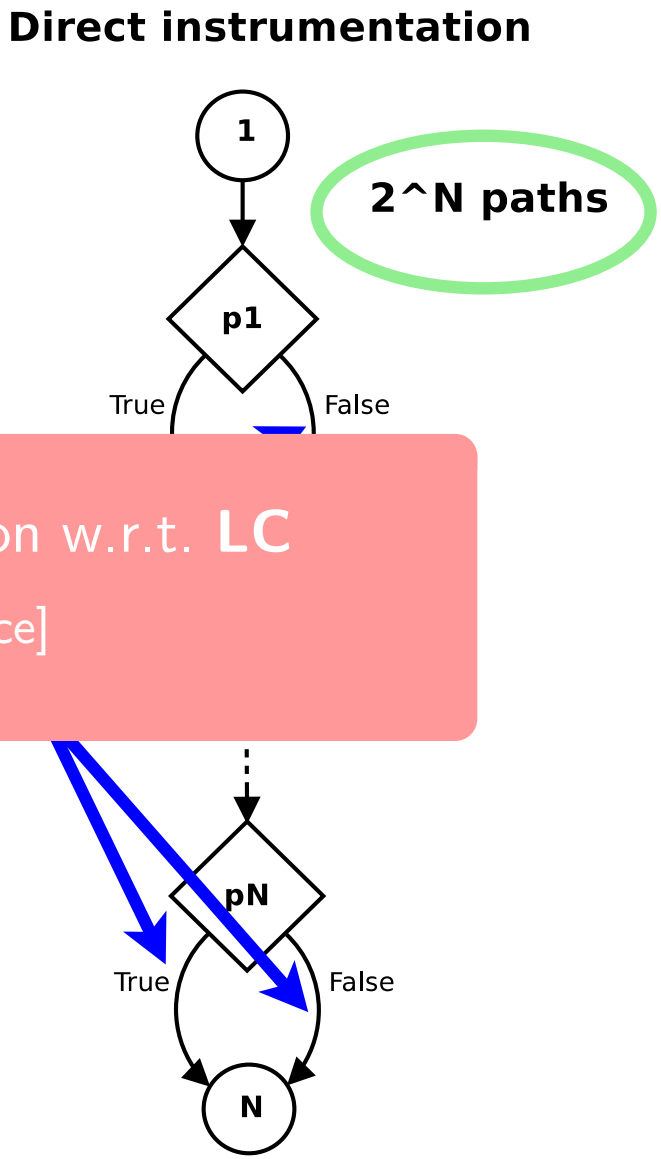
Non-tightness 2

× 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 is not good enough



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

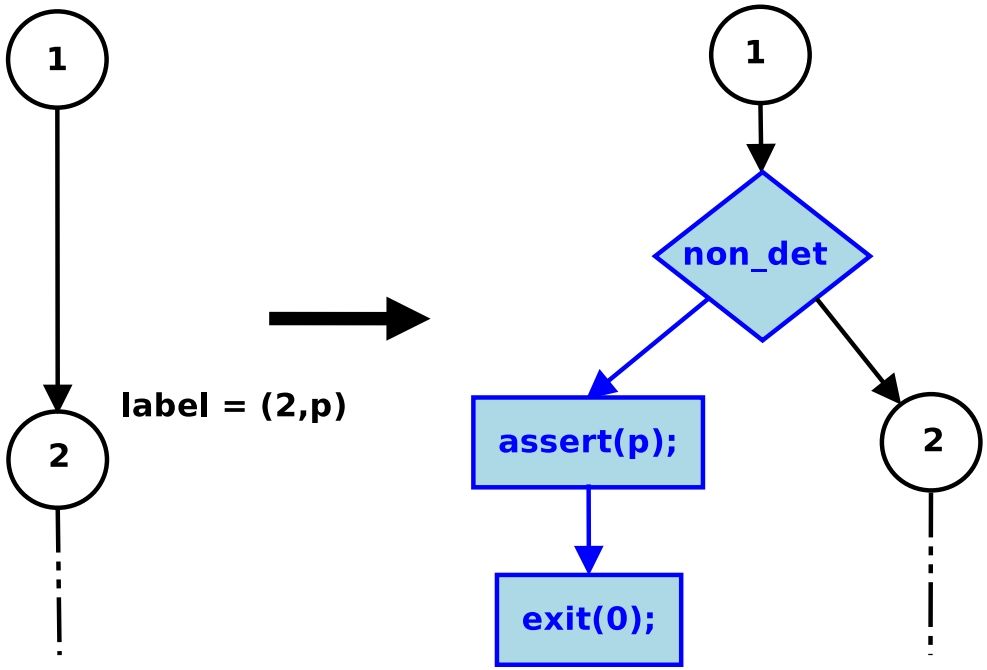
✗ dramatic overhead [theory & practice]

Our approach

The DSE* algorithm

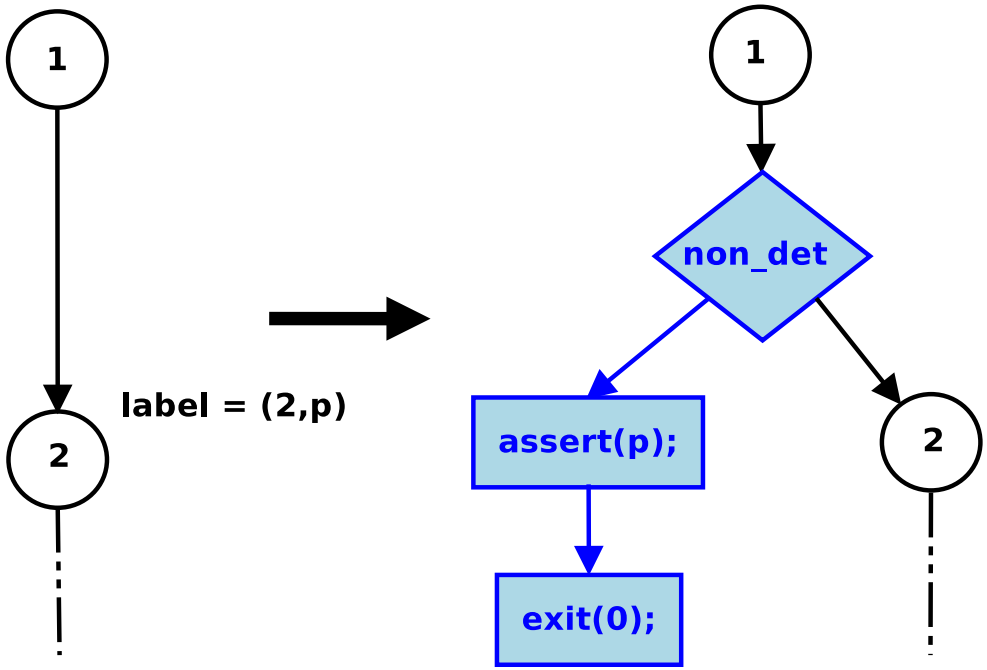
- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in black-box

DSE* : Tight Instrumentation



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

DSE* : Tight Instrumentation

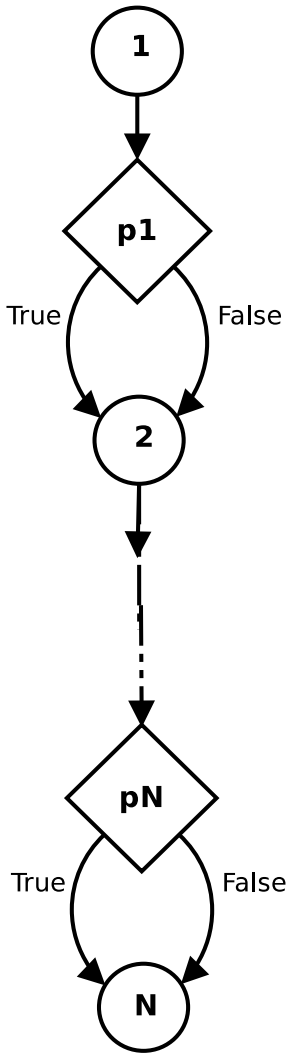


Covering label l $\Leftrightarrow$ Covering `exit(0)`

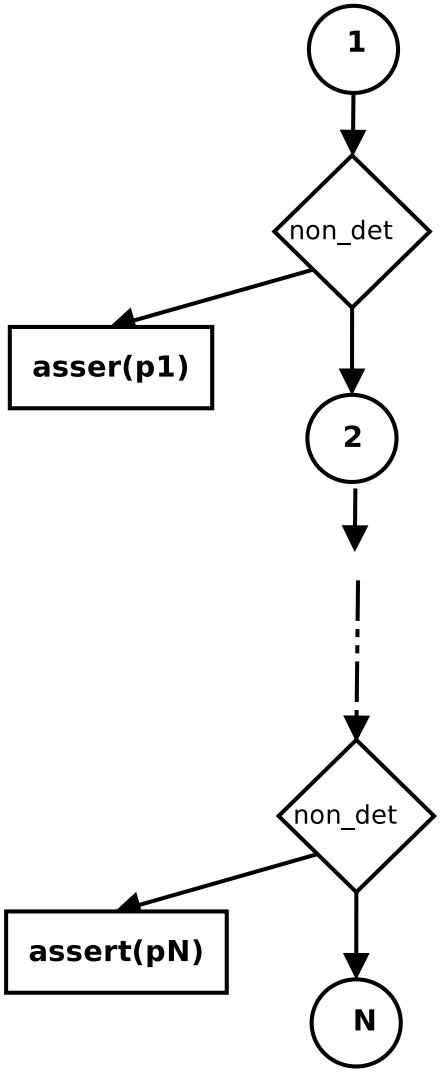
✓ sound & complete instrumentation w.r.t. **LC**

DSE* : Tight Instrumentation (2)

Direct instrumentation

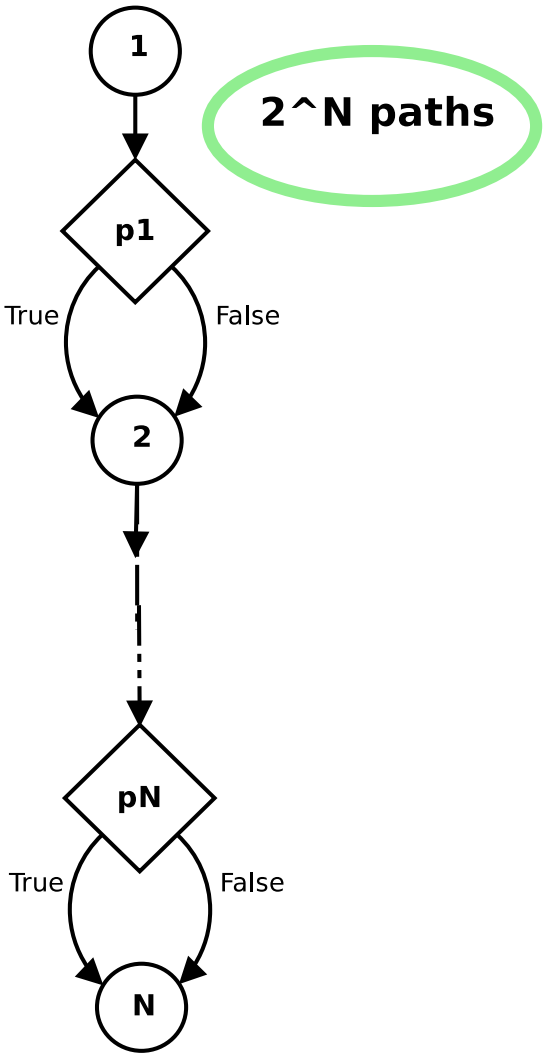


Tight Instrumentation

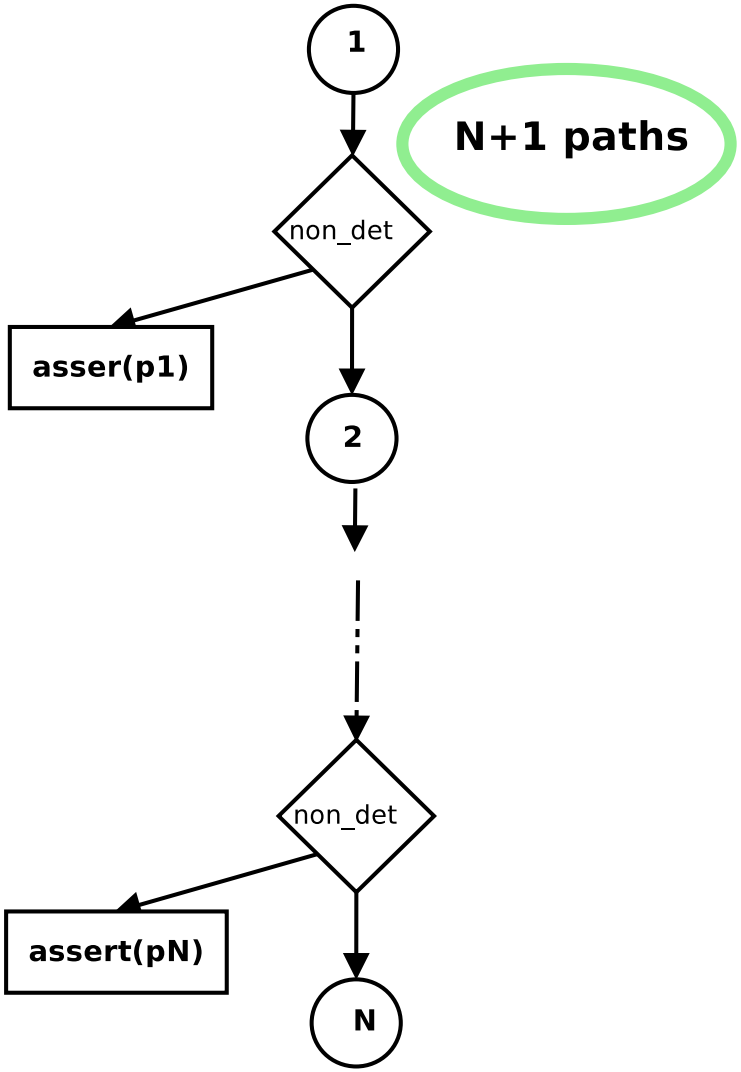


DSE* : Tight Instrumentation (2)

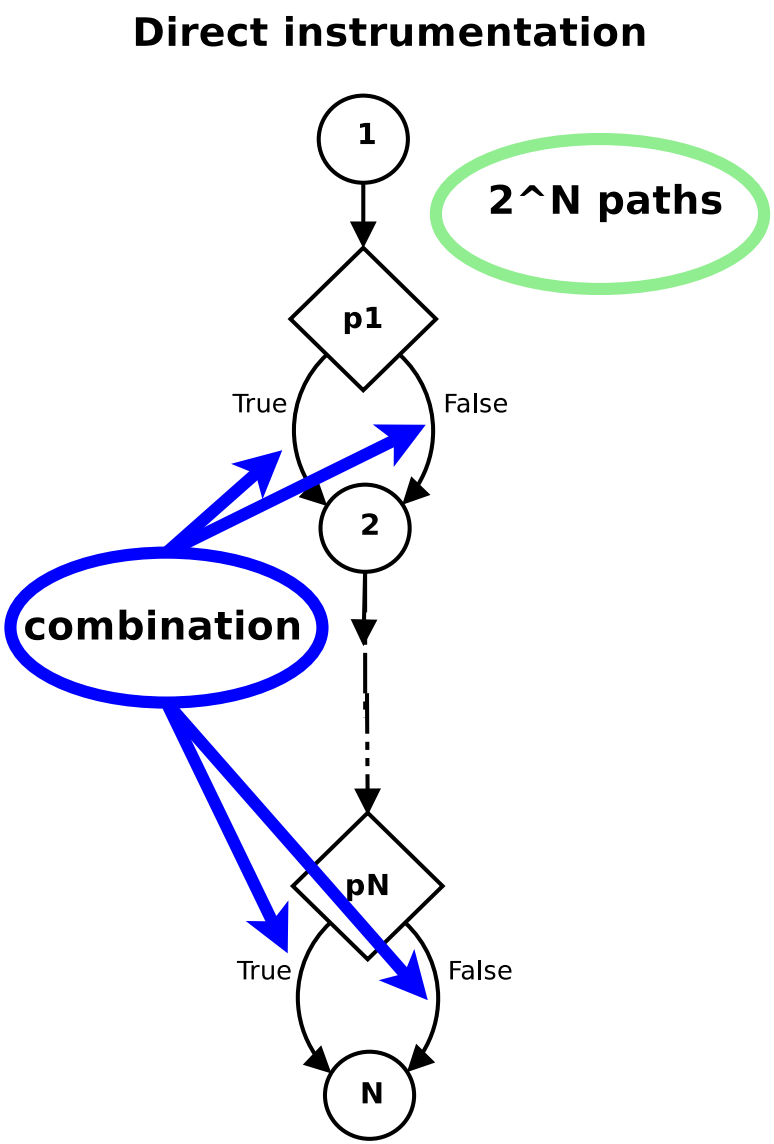
Direct instrumentation

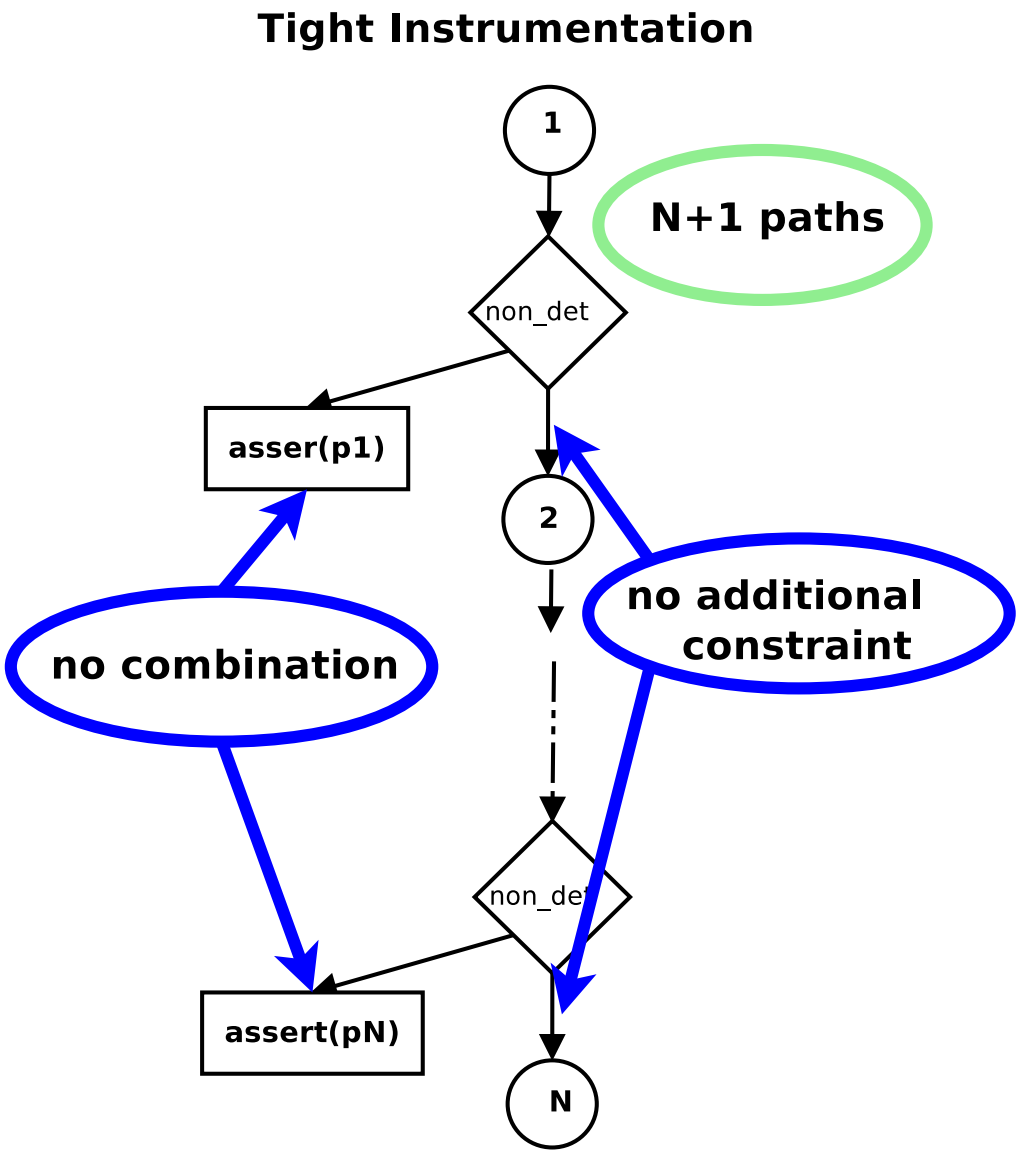


Tight Instrumentation

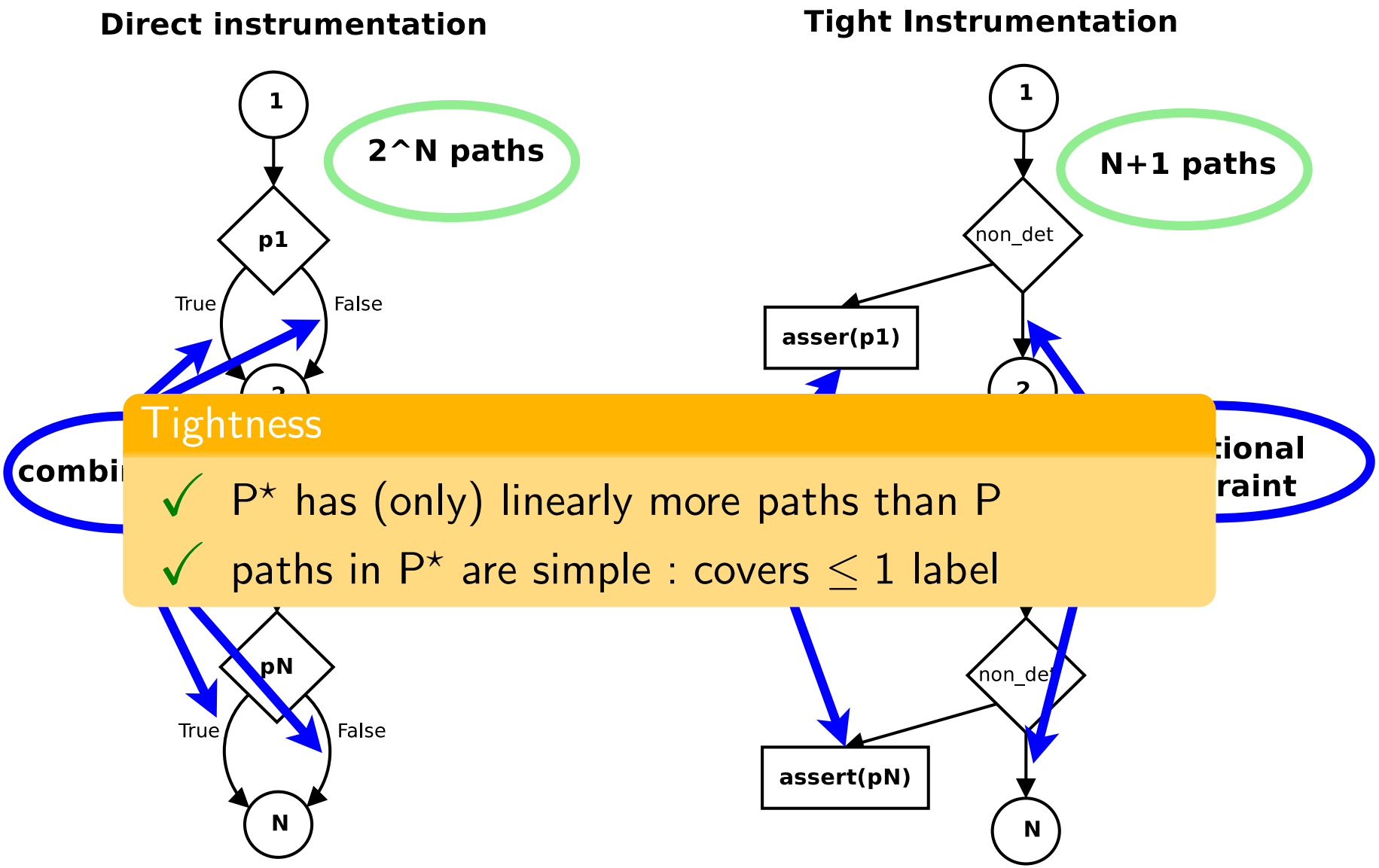


DSE* : Tight Instrumentation (2)





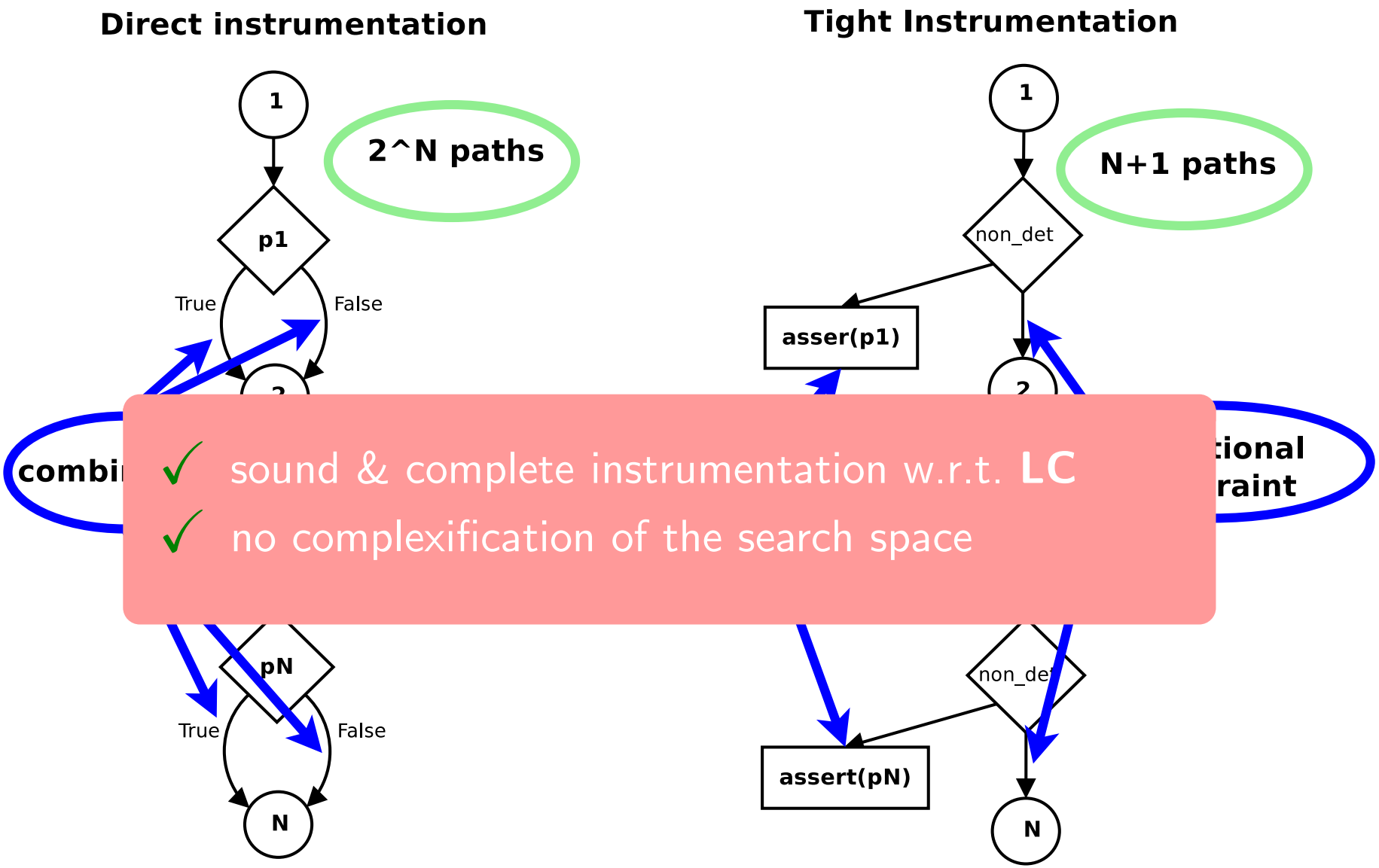
DSE* : Tight Instrumentation (2)



Tightness

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

DSE* : Tight Instrumentation (2)



- ✓ 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
- 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
- burden DSE with "useless" paths w.r.t. **LC**

Solution : Iterative Label Deletion

- keep a cover status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates cover 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
- burden DSE with "useless" paths w.r.t. **LC**

Solution : Iterative Label Deletion

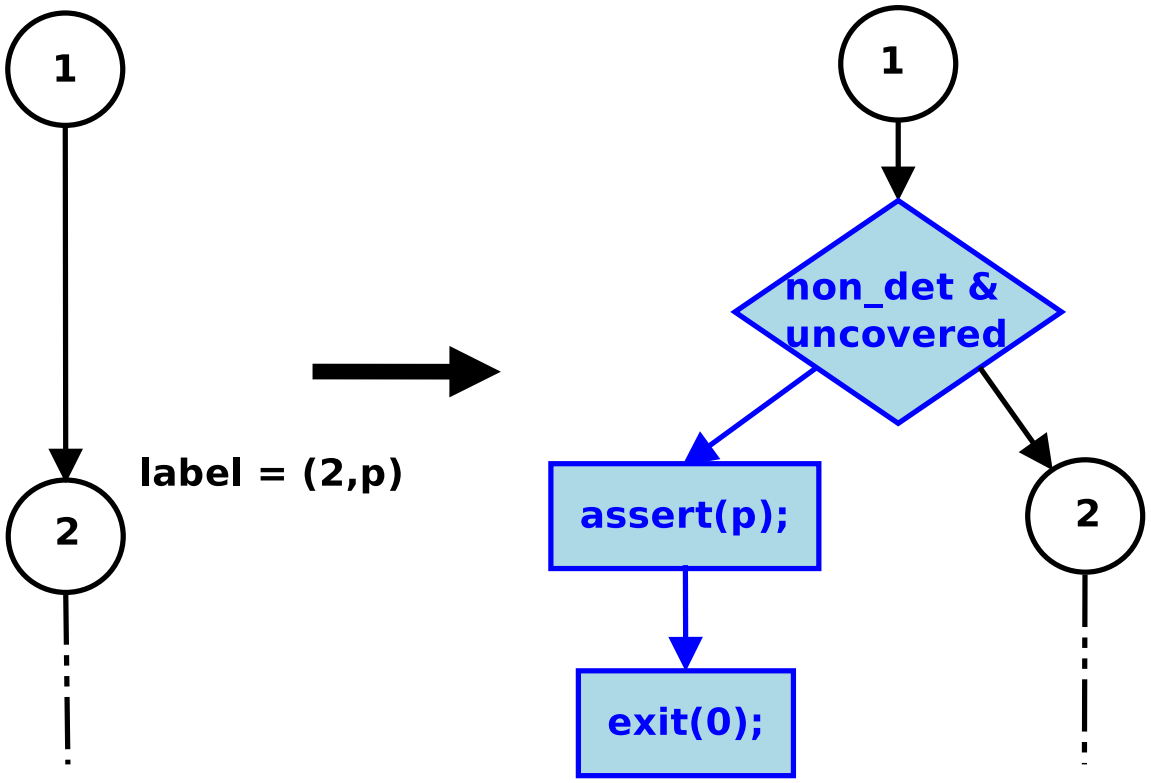
- keep a cover status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates cover status [truly requires DSE]

Implementation

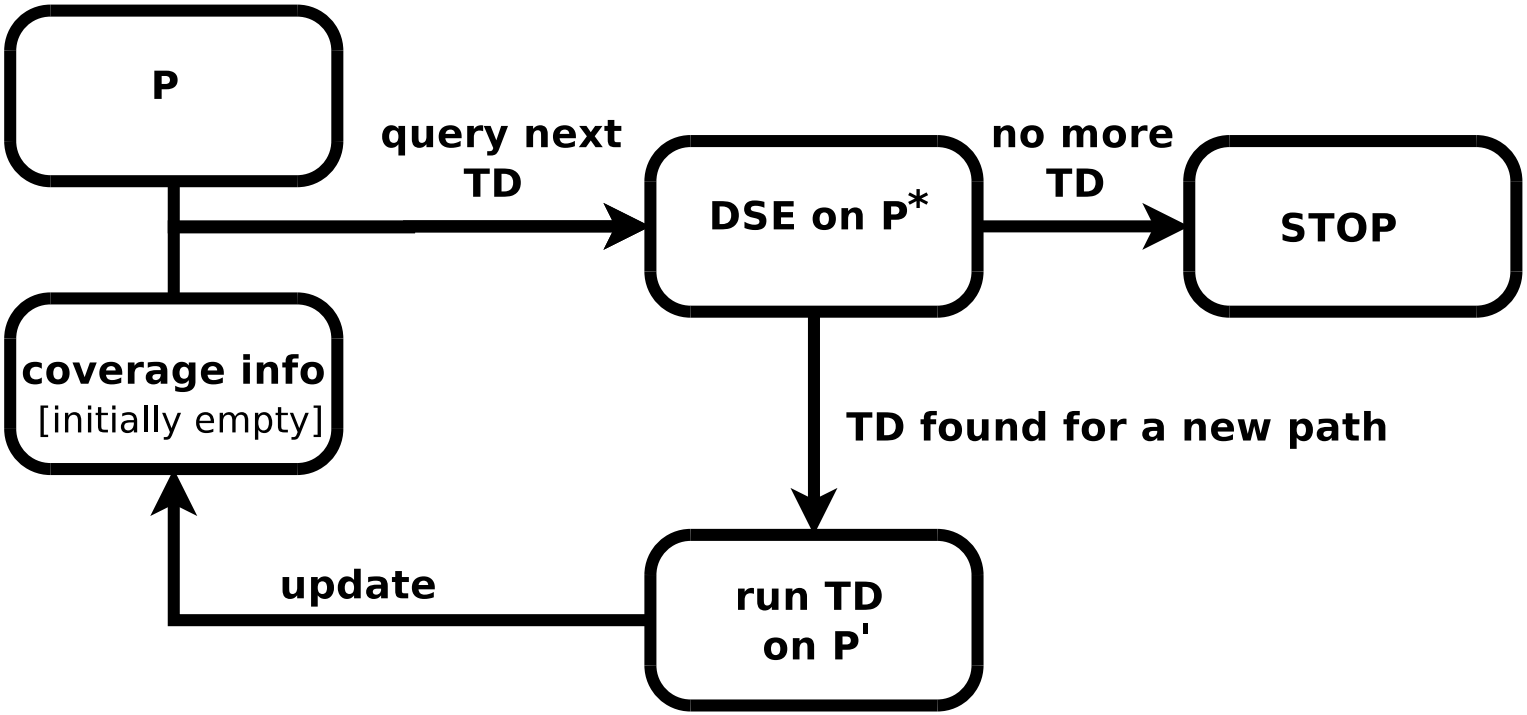
- symbolic part : a slight modification of $P^\star$
- dynamic part : a slight modification of P'

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

DSE* : Iterative Label Deletion (2)



DSE* : Iterative Label Deletion (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

Summary

The DSE* algorithm

- tion"

Goals

- ✓ OBJ1 : generic specification mechanism for coverage criteria
- ✓ OBJ2 : efficient integration into DSE

Outline

- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion

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

- 12 programs taken from standard DSE benchmarks (Siemens, Verisec, MediaBench) [beware : small programs]
- 3 coverage criteria : **CC**, **MCC**, **WM**
  [uncoverable labels not discarded]

Experiments (2)

Results

- DSE' : 4 TO, max overhead 122x [excluding TO]
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd_5-*wm* : 94s vs TO [1h30]
- also : DSE* achieves very high **LC**-coverage [$>$ 90% on 28/36]

Experiments (2)

Results

- DSE' : 4 TO, max overhead 122x [excluding TO]
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd_5-*wm* : 94s vs TO [1h30]
- also : DSE* achieves very high **LC**-coverage [$>$ 90% on 28/36]

Experiments (2)

Results

- DSE' : 4 TO, max overhead 122x [excluding TO]
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd_5-*wm* : 94s vs TO [1h30]
- also : DSE* achieves very high **LC**-coverage [$>$ 90% on 28/36]

Experiments (2)

Results

- DSE' : 4 TO, max overhead 122x [excluding TO]
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd_5-*wm* : 94s vs TO [1h30]
- also : DSE* achieves very high **LC**-coverage [$>$ 90% on 28/36]

Conclusion

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

A few results

			DSE	DSE'	DSE*
utf8-5 108 loc	wm 84 /	#paths time cover	680 2s	**11,111** 40s 82/84	**743** 8.1s 82/84
utf8-7 108 loc	wm 84 /	#paths time cover	3,069 5.8s	**81,133** **576s** 82/84	**3,265** **35s** 82/84
tcas 124 loc	wm 111 /	#paths time cover	4,420 5.6s	**300,213** **662s** 101/111	**6,014** **27s** 101/111
replace 100 loc	wm 79 /	#paths time cover	866 2s	**87,498** **245s** 70/79	**2,347** **14s** 70/79
get_tag-6 240 loc	cc 20 /	#paths time cover	76,456 3,011s	**TO**	76,468 **1,512s** 20/20
	wm 47 /	#paths time cover	76,456 3,011s	**TO**	76,481 **1,463s** 44/47
gd-5	wm 63 /	#paths time cover	14,516 50s	**TO**	14,607 **94s** 62/63
gd-6	wm 63 /	#paths time cover	107,410 3,740s	**TO**	107,521 **2,232s** 63/63

Outline

- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion

Conclusion

Goal = extend DSE to 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]

Conclusion

Goal = extend DSE to 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]

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

✓ very powerful approach to (white box) test generation

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

Conclusion

Goal = extend DSE to 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]

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

✓ very powerful approach to (white box) test generation

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

✗ support only basic coverage criteria

Conclusion

Goal = extend DSE to 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]

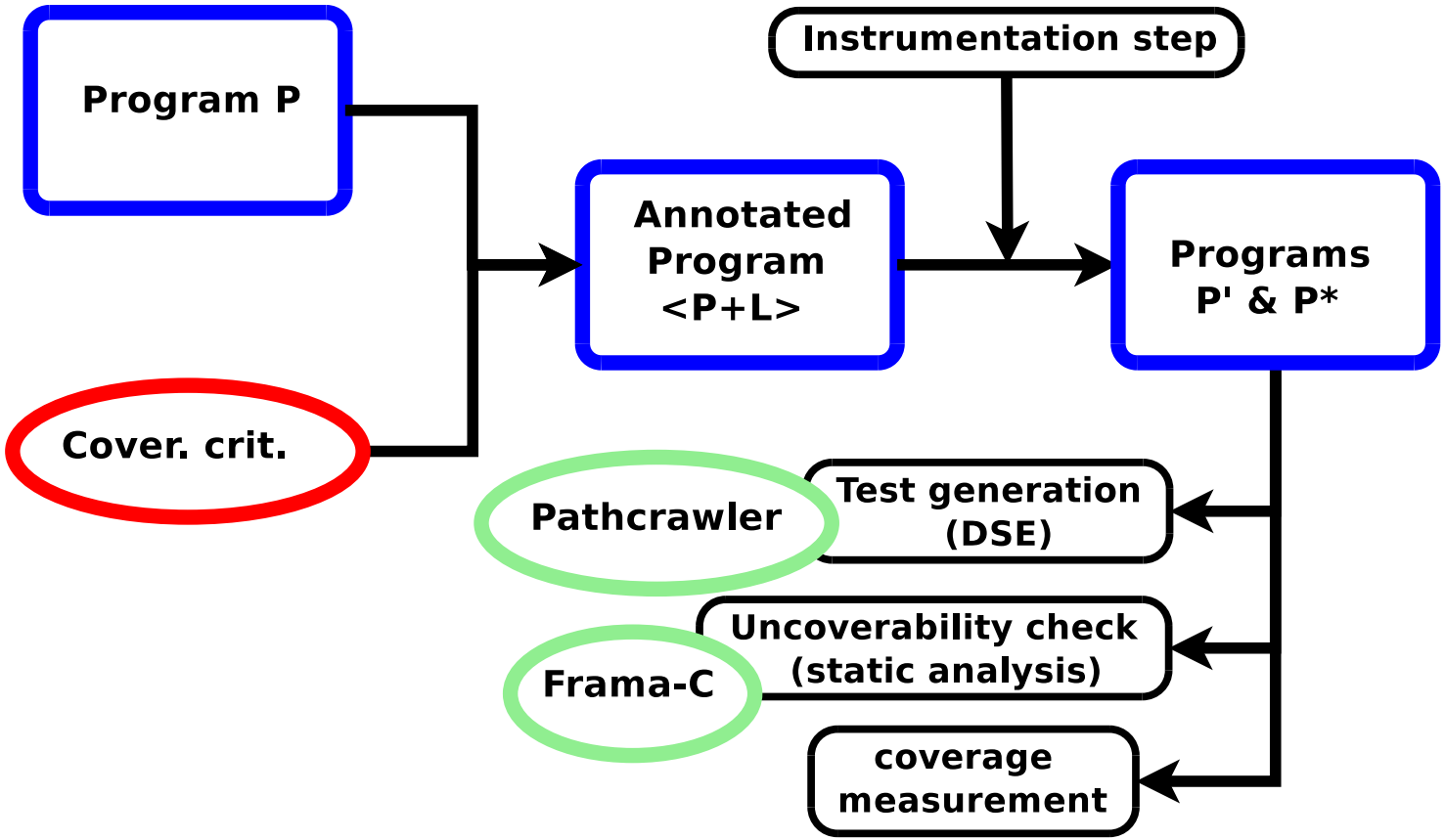
Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, . . .]

✓ very powerful approach to (white box) test generation

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

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

Coming soon


Program P
Instrumentation step
Annotated Program <P+L>
Programs P' & P*
Cover. crit.
Pathcrawler
Test generation (DSE)
Uncoverability check (static analysis)
Frama-C
coverage measurement