Enhancing Symbolic Execution for Coverage-Oriented Testing

Sébastien Bardin, Nikolai Kosmatov, Mickaël Delahaye

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

Context : white-box software testing

Testing process

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

Coverage criteria [decision, mcdc, mutants, etc.] play a major role

- definition = systematic way of deriving test requirements
- generate tests, decide when to stop, assess quality of testing
- beware : infeasible test requirements
  [waste generation effort, imprecise coverage ratios]
- beware : lots of different coverage criteria

Context : 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

Context : 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

Symbolic Execution [King 70's]

- consider a program $\mathtt{P}$ on input $\mathtt{v}$, and a given path σ
- a path predicate φ_σ for σ is a formula s.t.

$$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \sigma$$

- can be used for bounded-path testing !
- old idea, recent renew interest [requires powerful solvers]

Context : 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

Symbolic Execution [King 70's]

- consider a program $\mathtt{P}$ on input $\mathtt{v}$, and a given path σ
- a path predicate φ_σ for σ is a formula s.t.

$$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \sigma$$

- 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]

Context : Dynamic Symbolic Execution (2)

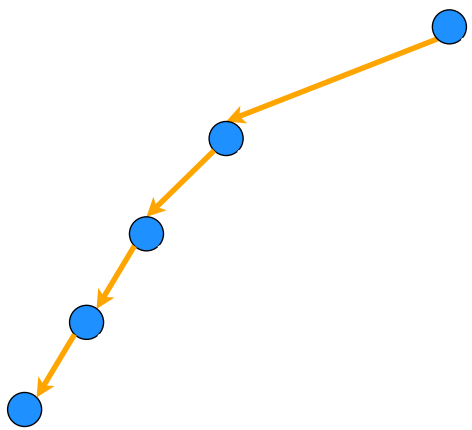
input : a program P
output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- 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, $\sigma >$
- loop until no more path to cover

Context : Dynamic Symbolic Execution (2)

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

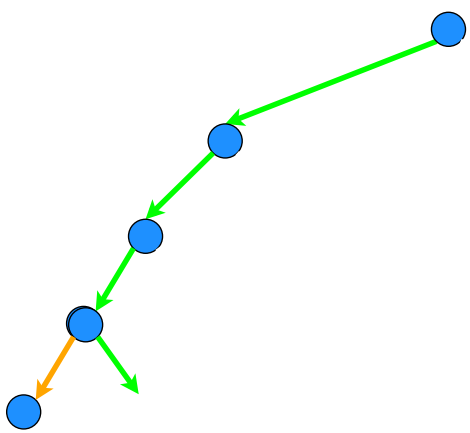
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Context : Dynamic Symbolic Execution (2)

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

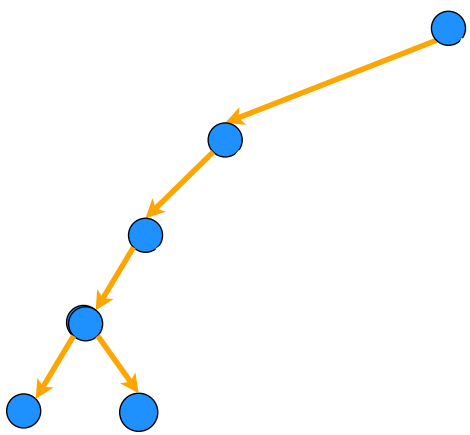
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Context : Dynamic Symbolic Execution (2)

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

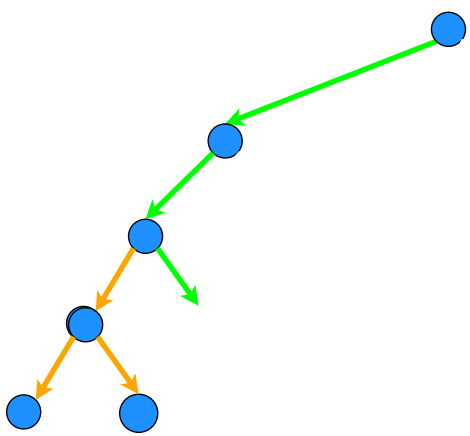
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Context : Dynamic Symbolic Execution (2)

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

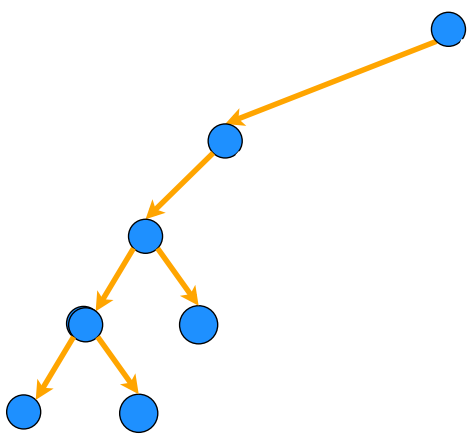
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Context : Dynamic Symbolic Execution (2)

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

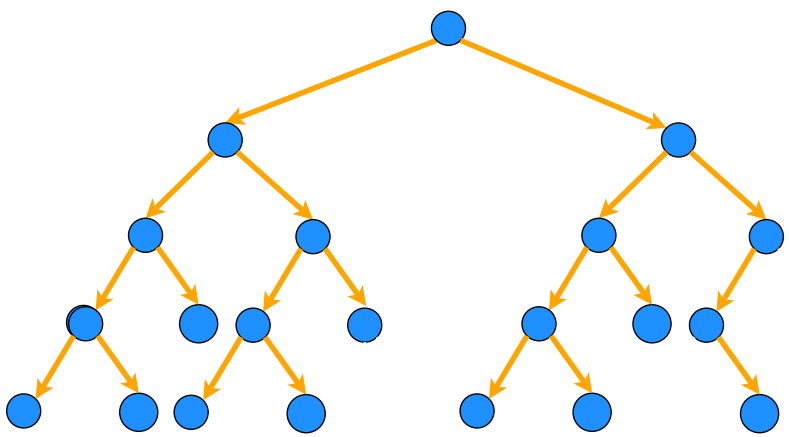
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover



Context : 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

DSE is GREAT for automating structural testing

- ✓ very powerful approach to (white box) test generation
- ✓ many tools and many successful case-studies since mid 2000's

The problem

DSE is GREAT for automating structural testing

- ✓ very powerful approach to (white box) test generation
- ✓ many tools and many successful case-studies since mid 2000's

Yet, no real support for structural coverage criteria

[except path coverage and branch coverage]

Would be useful :

- when required to produce tests achieving some criterion
- for producing "good" tests for an external oracle
  [functional correctness, security, performance, etc.]

Our goals and results

Goals : extend DSE to a large set of structural coverage criteria

- support these criteria in a unified way
- support these criteria in an efficient way
- detect (some) infeasible test requirements

Our goals and results

Goals : extend DSE to a large set of structural coverage criteria

- support these criteria in a unified way
- support these criteria in an efficient way
- detect (some) infeasible test requirements

Results

- ✓ generic low-level encoding of coverage criteria [ICST 14]
- ✓ efficient variant of DSE for coverage criteria [ICST 14]
- ✓ sound and quasi-complete detection of infeasibility [ICST 15]
- more complete experiments, **GACC** criterion, optimization of DSE through static analysis

Outline

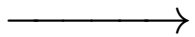
- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

Focus : Labels

- Annotate programs with **labels**
  - predicate attached to a specific program instruction
- Label (loc, φ) is covered if a test execution
  - reaches the instruction at loc
  - satisfies the predicate φ
- **Good for us**
  - can easily encode a large class of coverage criteria [see after]
  - in the scope of standard program analysis techniques

Simulation of standard coverage criteria

```
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

```
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

```
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;
```

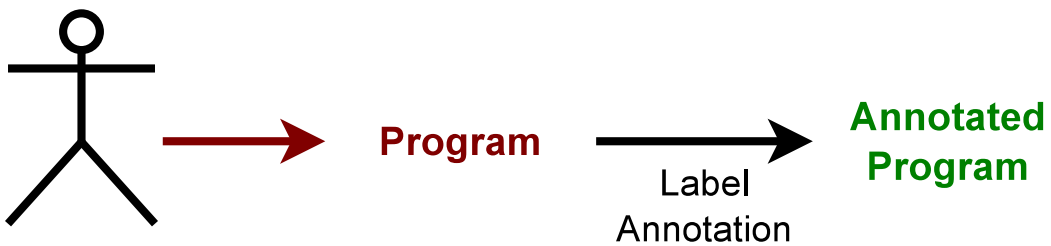
Multiple-Condition Coverage (**MCC**)

Simulation of standard coverage criteria

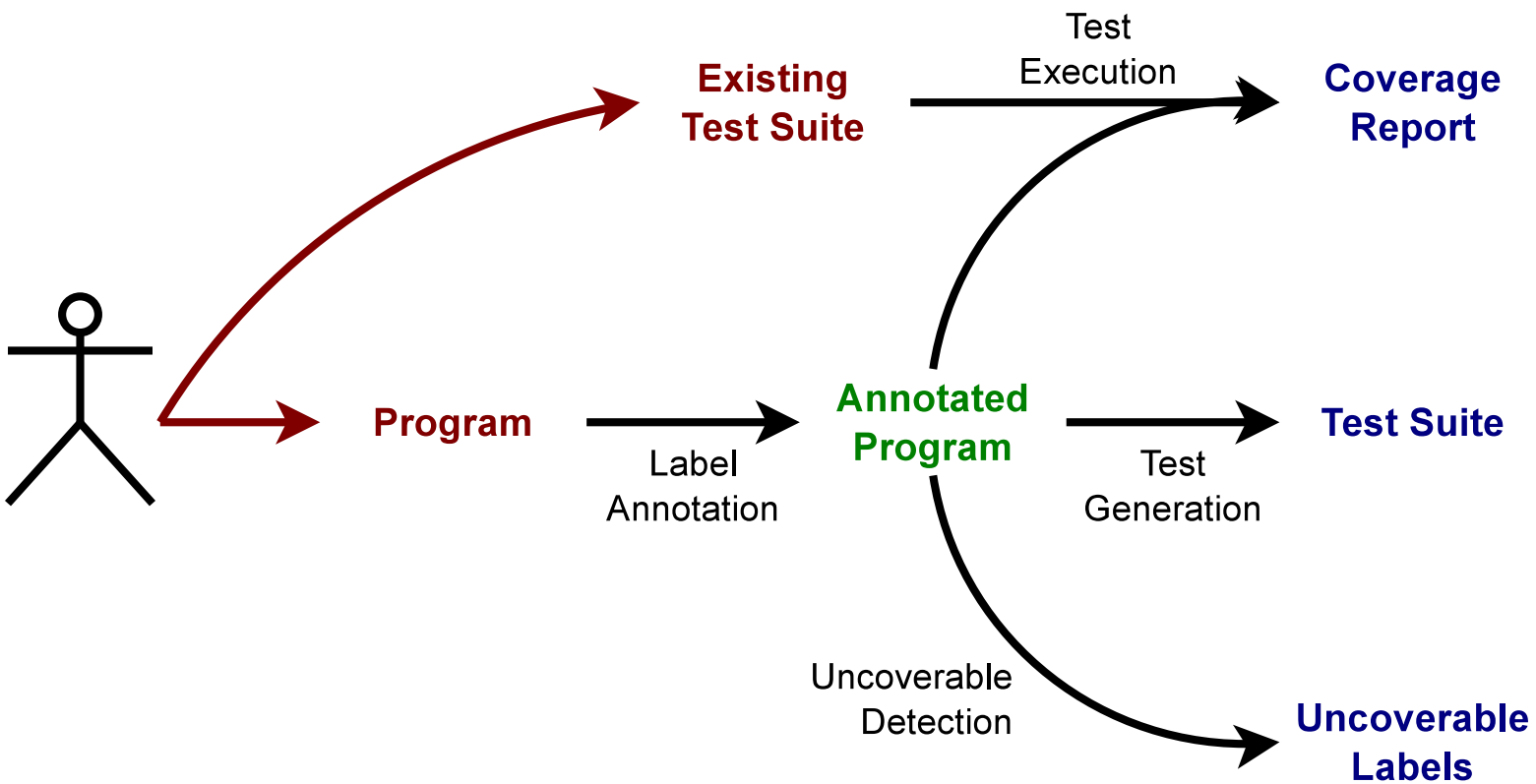
OBJ : generic specification mechanism for coverage criteria ✓

- **IC**, **DC**, **FC**, **CC**, **MCC**, **GACC**
- **large part of Weak Mutations**
- Input Domain Partition
- Run-Time Error

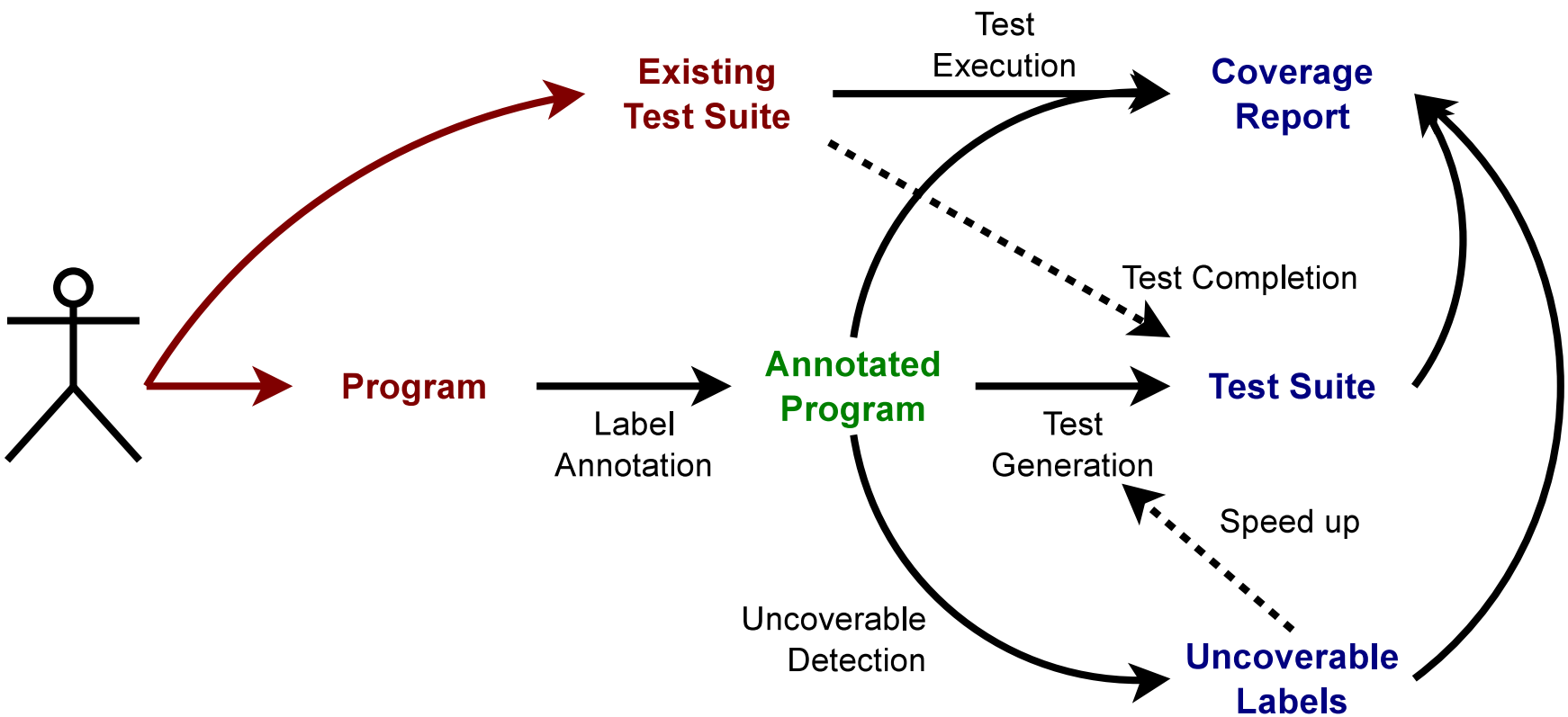
LTest overview



LTest overview



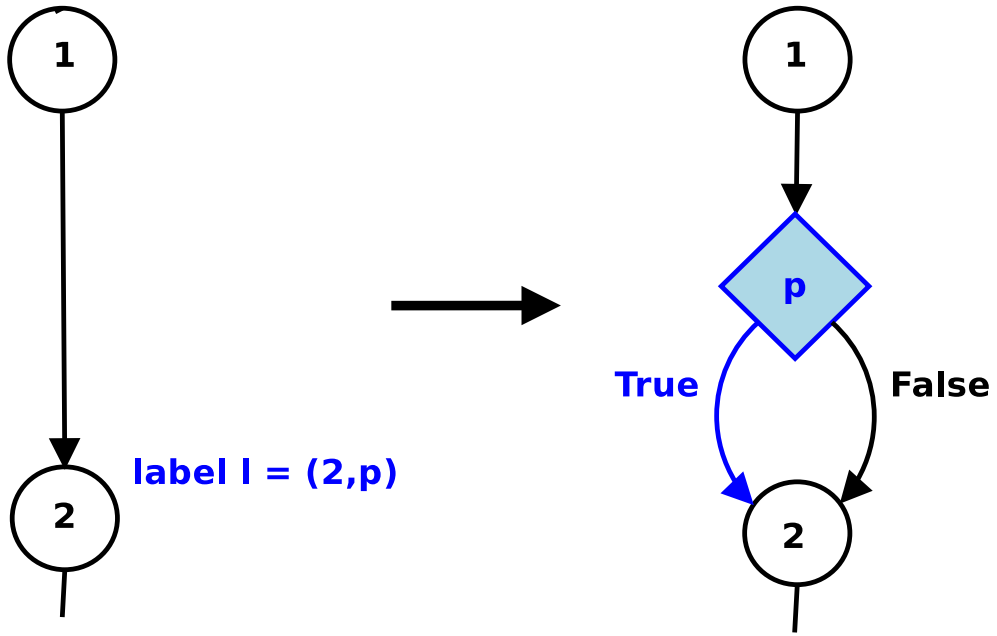
LTest overview



Outline

- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

Direct instrumentation

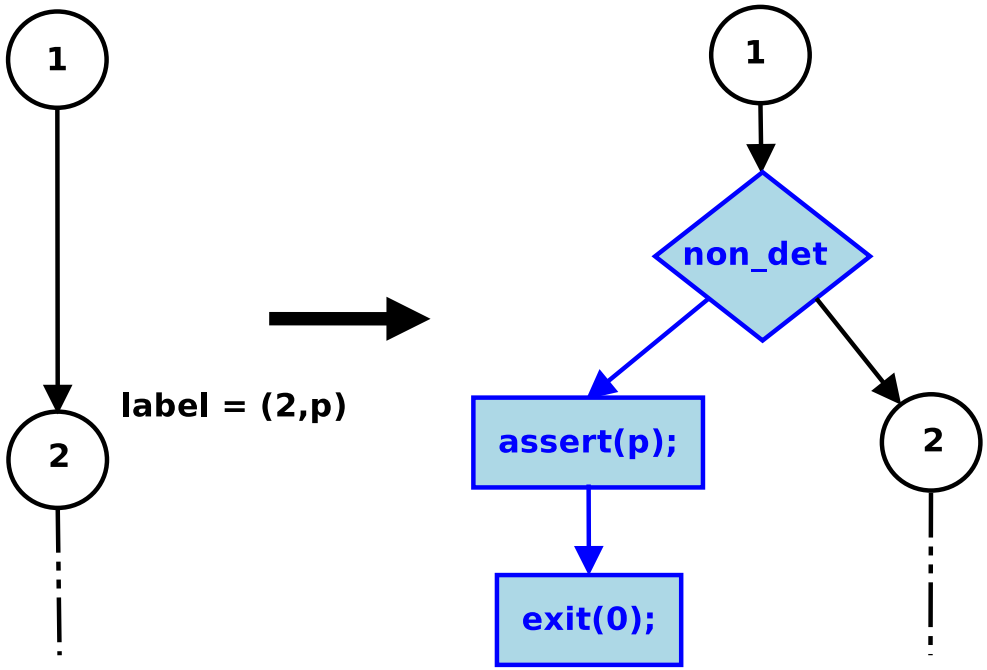


Covering label l $\Leftrightarrow$ Covering branch `True`

✓ sound & complete instrumentation

✗ dramatic overhead [theory & practice]

DSE* : Tight Instrumentation

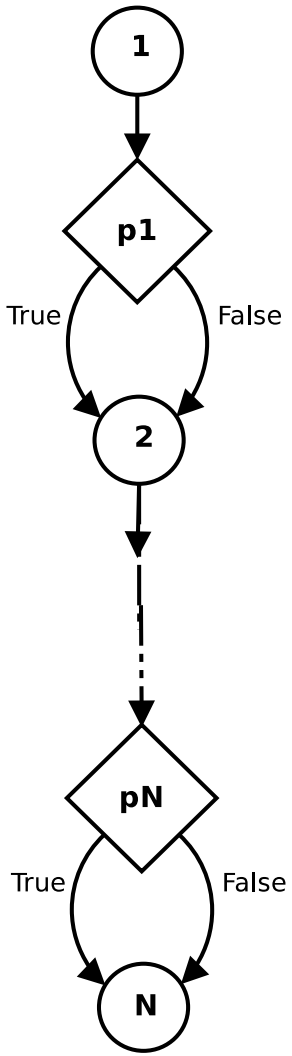


Covering label l $\Leftrightarrow$ Covering `exit(0)`

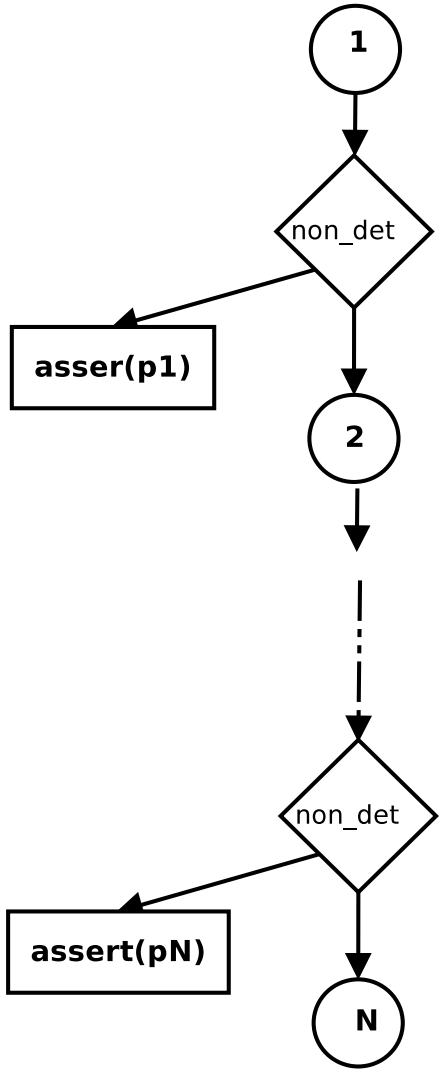
- ✓ sound & complete instrumentation
- ✓ no complexification of the search space

DSE* : Tight Instrumentation (2)

Direct instrumentation

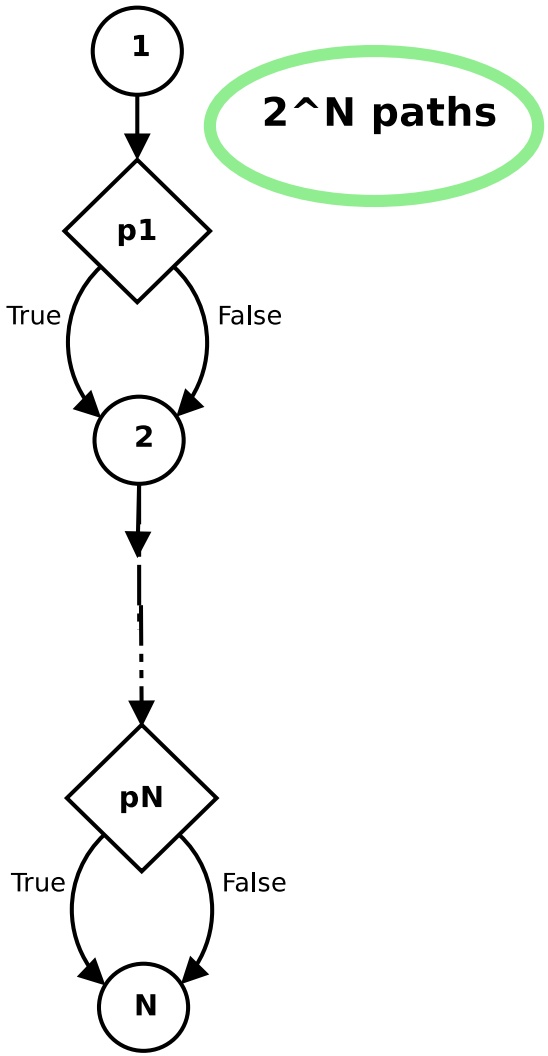


Tight Instrumentation

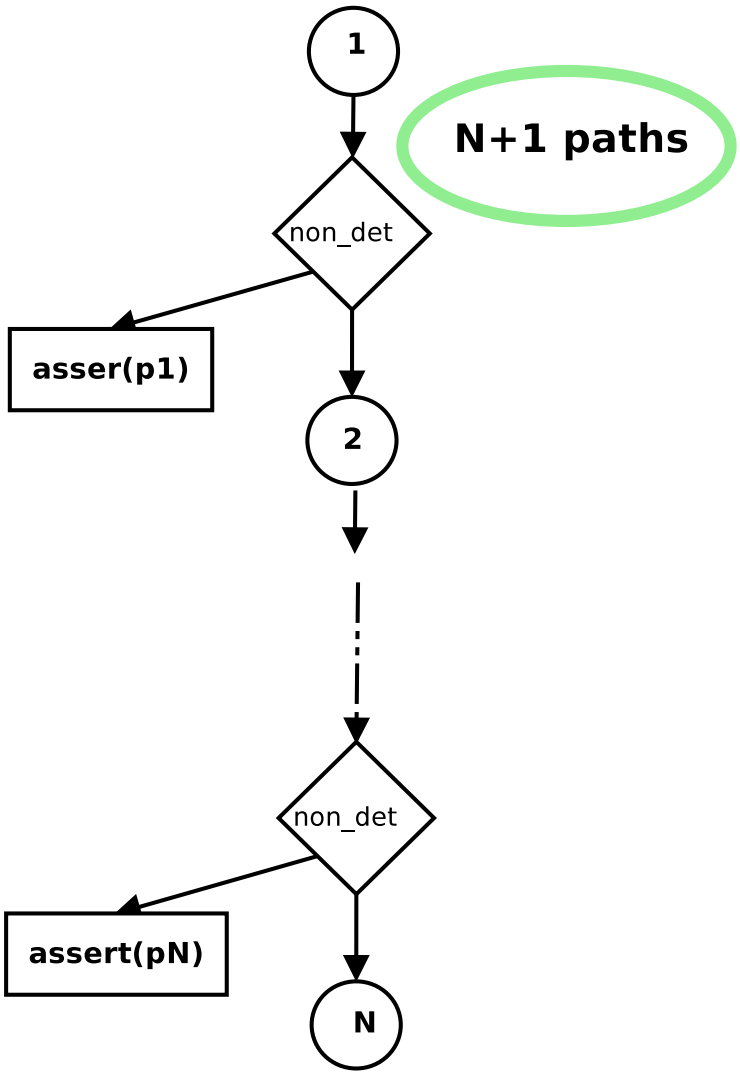


DSE* : Tight Instrumentation (2)

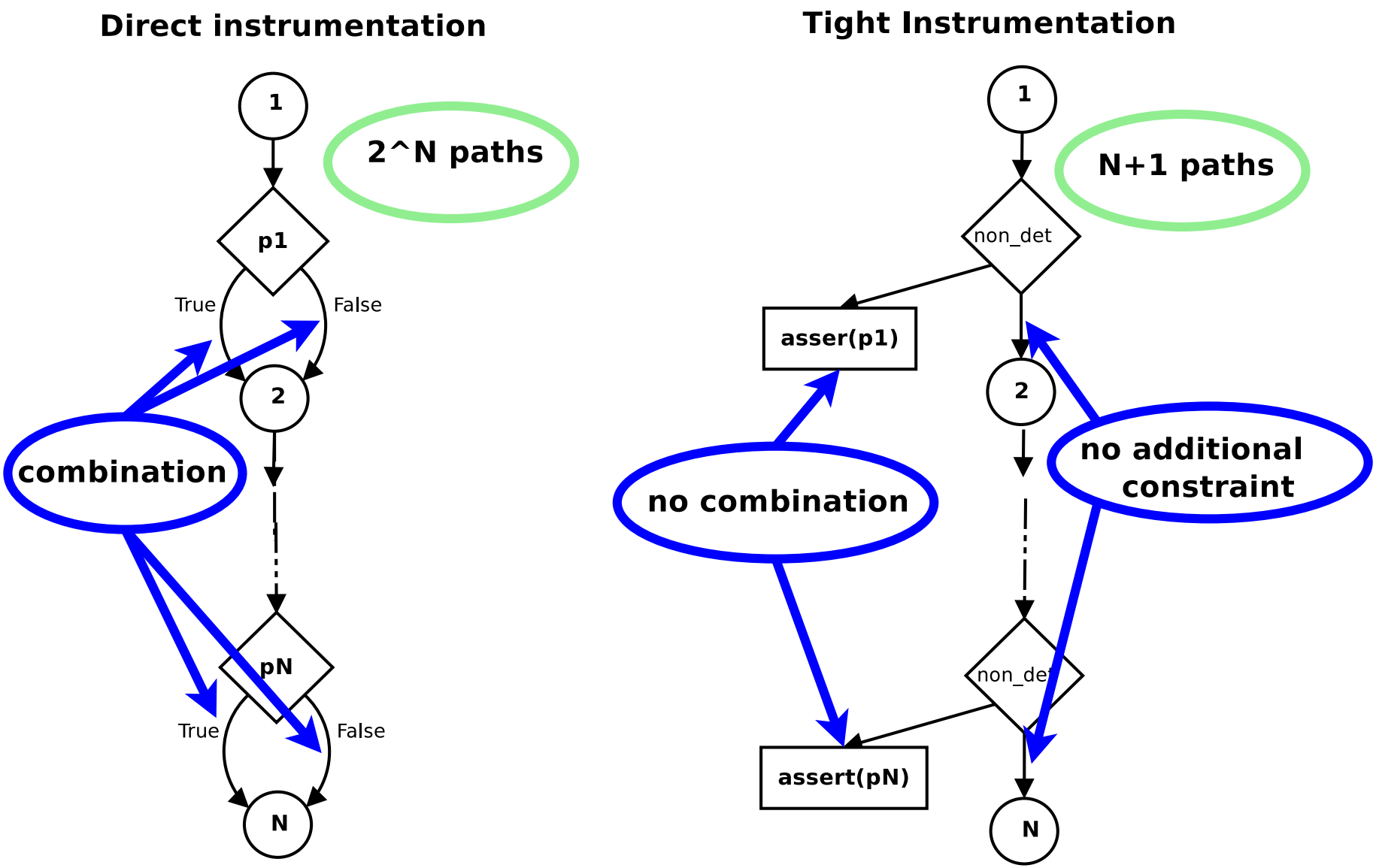
Direct instrumentation



Tight Instrumentation



DSE* : Tight Instrumentation (2)



Experiments

Benchmark : Standard (test generation) benchmarks [Siemens, Verisec, Mediabench]

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

Performance overhead

	DSE	DSE'	DSE*
Min	×1	×1.02	×0.49
Median	×1	×1.79	×1.37
Max	×1	**×122.50**	**×7.15**
Mean	×1	**×20.29**	**×2.15**
Timeouts	0	**5** *	**0**

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

Experiments

Benchmark : Standard (test generation) benchmarks [Siemens, Verisec, Mediabench]

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

Coverage

	Random	DSE	DSE*
Min	37%	61%	62%
Median	63%	90%	**95%**
Max	100%	100%	100%
Mean	70%	87%	**90%**

vs DSE : +39% coverage on some examples

Experiments

Benchmark : Standard (test generation) benchmarks [Siemens, Verisec, Mediabench]

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

Conclusion

- DSE* performs significantly better than DSE'
- The overhead of handling labels is kept reasonable
- high coverage, better than DSE

Outline

- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

Infeasibility detection

Basic ideas

- rely on sound verification methods
  - . label (loc, φ) infeasible $\Leftrightarrow$ assertion $(loc, \neg\varphi)$ invariant
- grey-box combination of existing approaches [VA $\oplus$ WP]

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

- 12 programs (50-300 loc), 3 criteria (**CC**, **MCC**, **WM**)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements, 121 infeasible ones

Detection power

	#Lab	#Inf	VA ⊕ WP	
			#d	%d
Total	1,270	121	118	**98%**
Min		0	2	**67%**
Max		29	29	100%
Mean		4.7	4.5	**95%**

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels

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

Impact on test generation

report more accurate coverage ratio

	Coverage ratio reported by DSE*		
Detection method	**None**	**VA ⊕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

Impact on test generation

optimisation : speedup test generation : take care !

Ideal speedup

	DSE*-OPT vs DSE*
Min.	0.96×
Max.	592.54×
Mean	49.04×

in practice

		DSE*-OPT vs DSE*
RT(1s) +LUncov +DSE*	**Min**	0.1x
	Max	**55.4x**
	Mean	**3.8x**

RT : random testing
Speedup wrt. DSE* alone

Impact on test generation

optimisation : speedup test generation : take care !

Ideal speedup

	DSE*-OPT vs DSE*
Min.	0.96×
Max.	592.54×
Mean	49.04×

in practice

		DSE*-OPT vs DSE*
RT(1s) +LUncov +DSE*	**Min**	0.1x
	Max	**55.4x**
	Mean	**3.8x**

RT : random testing
Speedup wrt. DSE* alone

Outline

- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

The GACC criterion

MCDC :

- coverage criterion used in aeronautics
- demanding, industrially relevant
- requires to show that any atomic condition c_i alone can influence its branching condition C

  . 2 tests t_1, t_2 for any such c_i

  . $t_1(c_i) \neq t_2(c_i)$, $t_1(C) \neq t_2(C)$, and $\forall j \neq i . t_1(C) = t_2(C)$

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

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

The GACC criterion

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
  - . 2 tests t_1, t_2 for any such c_i
  - . $t_1(c_i) \neq t_2(c_i)$ [nothing on $t_1(C), t_2(C)$]
  - . for both t_1, t_2, modifying c_i alones switches C

    [nothing on the $t_1(c_j), t_2(c_j)$]

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

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

The GACC criterion

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

. [no exact encoding is known for MCDC]

$$\varphi_i \triangleq \quad c_i = \mathbf{f} \ \wedge \ C(c_1, \ldots, c_{i-1}, \mathbf{t}, c_{i+1}, \ldots, c_n) \neq C(c_1, \ldots, c_{i-1}, \mathbf{f}, c_{i+1}, \ldots, c_n)$$

$$\varphi'_i \triangleq \quad c_i = \mathbf{t} \ \wedge \ C(c_1, \ldots, c_{i-1}, \mathbf{t}, c_{i+1}, \ldots, c_n) \neq C(c_1, \ldots, c_{i-1}, \mathbf{f}, c_{i+1}, \ldots, c_n)$$

The GACC criterion (2)

Performance overhead

	DSE'	DSE*	
		norm.	OPT
Min	1.44×	1.41×	1.38×
Med	3.76×	1.81×	1.44×
Max	130.79×	59.40×	3.14×
Mean	21.99×	10.55×	1.85×
Timeouts	1	0	0

Coverage

	Random	DSE	DSE*	
			norm.	OPT
Min	47%	62%	64%	72%
Med	55%	76%	88%	96%
Max	100%	100%	100%	100%
Mean	60%	78%	85%	91%

Outline

- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

Conclusion

Goals : extend DSE to a large set of structural coverage criteria

- ✓ generic low-level encoding of coverage criteria [ICST 14]
- ✓ efficient variant of DSE for coverage criteria [ICST 14]
- ✓ sound and quasi-complete detection of infeasibility [ICST 15]
- more complete experiments, **GACC** criterion, optimization of DSE through static analysis