Symbolic methods for efficient mutation testing (work in progress)

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle
Paris area, France

joint work with François Cheynier

Starting point

Dynamic Symbolic Execution [sage, pex, klee, pathcrawler, osmose, . . .]

- ✓ very powerful approach to (white box) test generation
- × but tools often target very basic coverage criteria (instructions, decisions)

Mutation coverage criterion

- ✓ very powerful coverage criterion
- × but lacks automation

Two main goals

Use symbolic methods to automate mutation testing

- test data generation
- equivalent mutant detection
- efficient mutant-coverage computation [not so easy with mutants]

Leverage DSE methods to more powerful coverage criteria

Approach : try to reuse symbolic tools in a blackbox manner

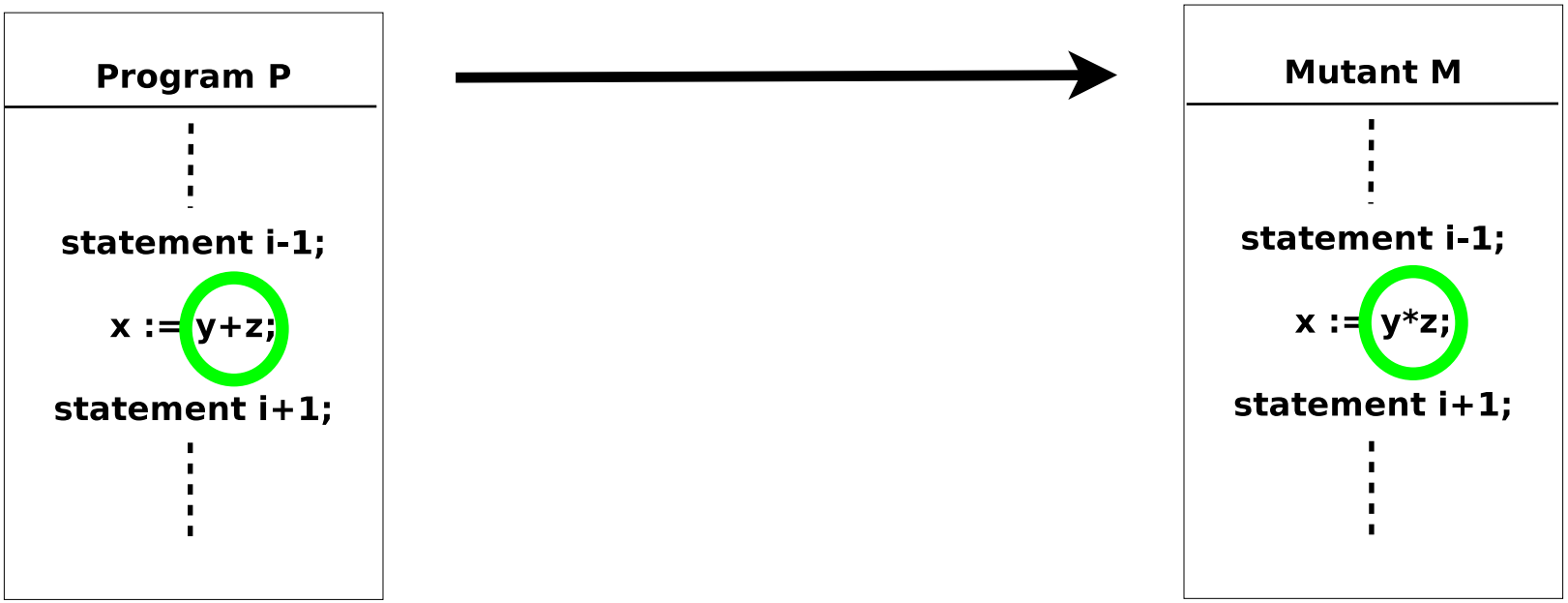
Current results

- identify which part of (weak) mutations can be efficiently automated
- leverage DSE to mutation testing (blackbox)
  - need careful instrumentation and incremental use of DSE
  - achieve high-level mutation coverage with reasonable overhead [on a few examples]
- explore other sources of automation :
  - very efficient mutant-coverage computation (low overhead)
  - effective detection of equivalent mutants

Outline

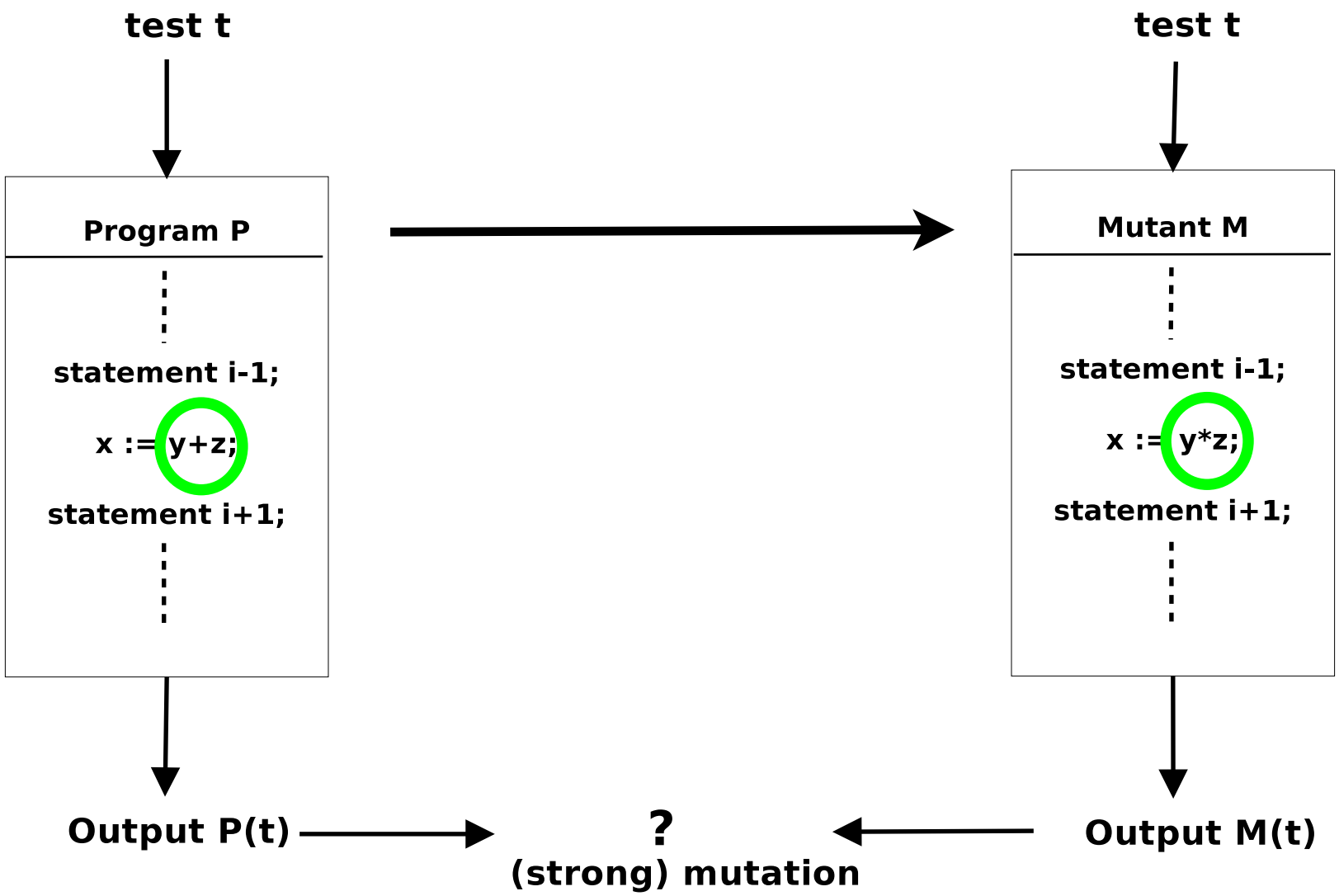
- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion

Mutation testing in a nutshell



- Mutant M = syntactic modification of program P

Mutation testing in a nutshell



- Mutant M = syntactic modification of program P
- Covering M = finding a DT t such that $P(t) \neq M(t)$

Difficulties for automation

Automatic mutant coverage

- have to reason on two different traces on two (slighty) different programs
- quite far from usual invariance / reachability problems

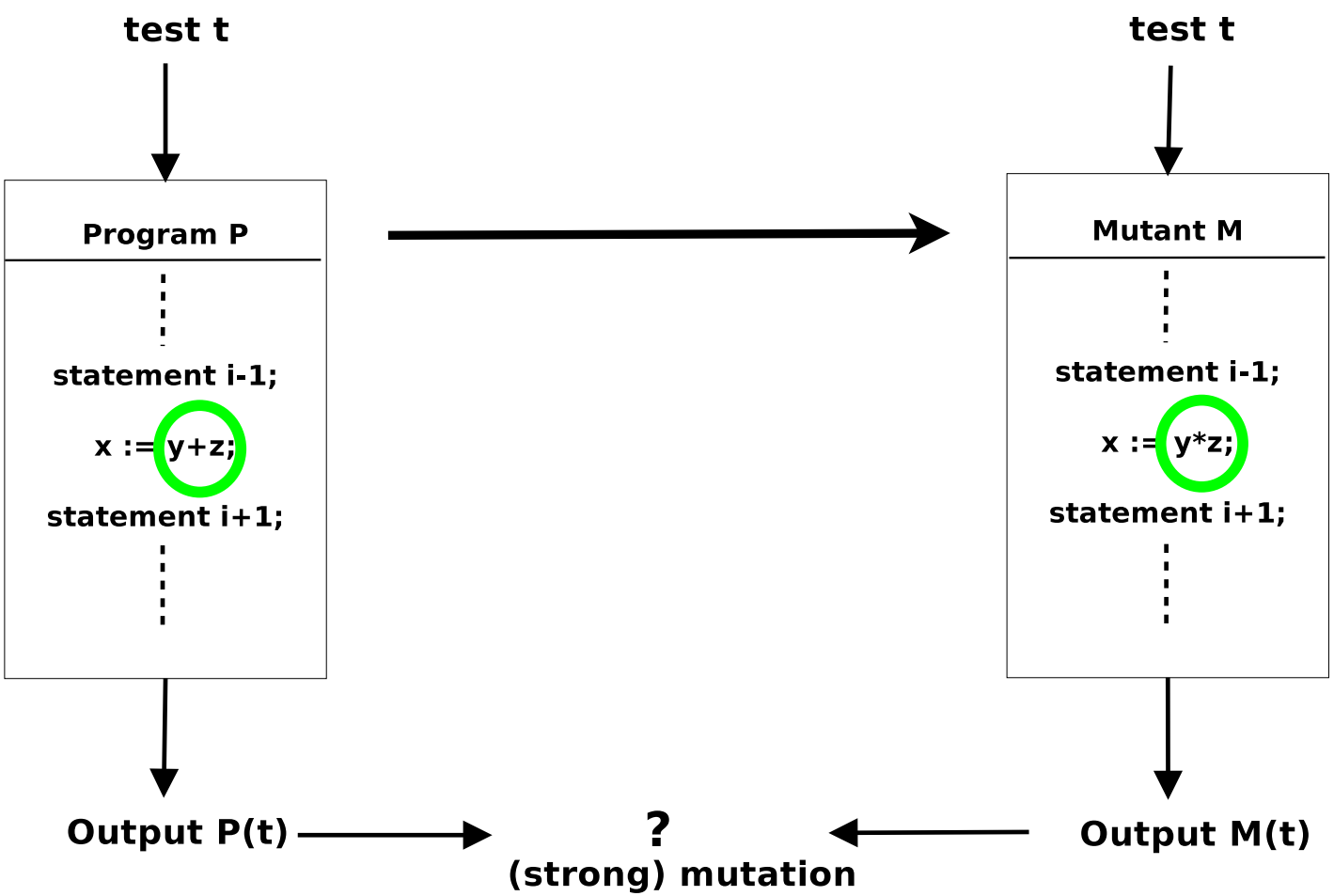
Efficient mutant-coverage computation (wrt. test set T)

- $|\mathcal{M}|$ compilations and $|T| \times (|\mathcal{M}| + 1)$ executions
  // standard D-coverage : 1 compilation, $|T|$ executions

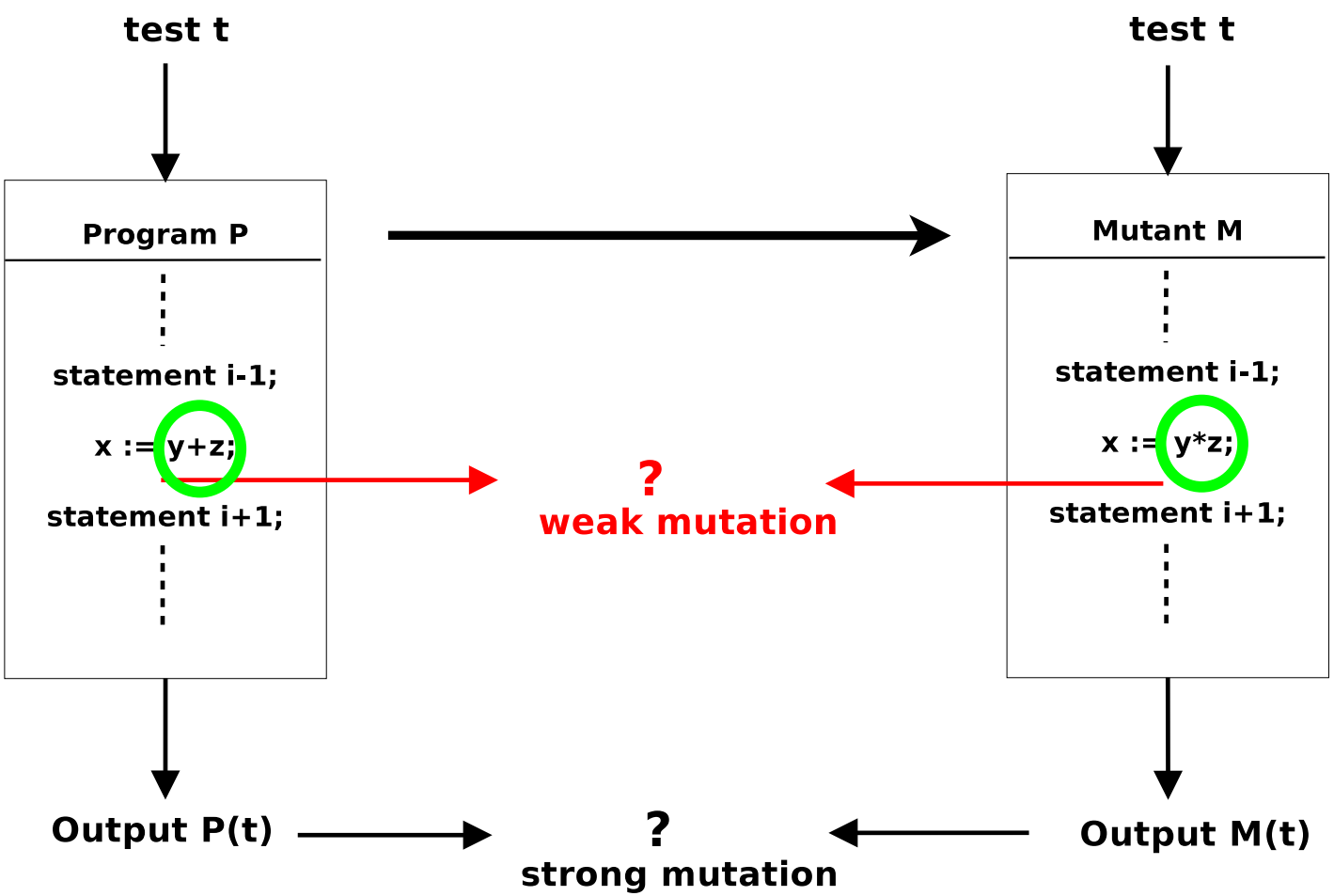
Problem of equivalent mutants

- cannot be distinguished from P
- waste ressources trying to cover them

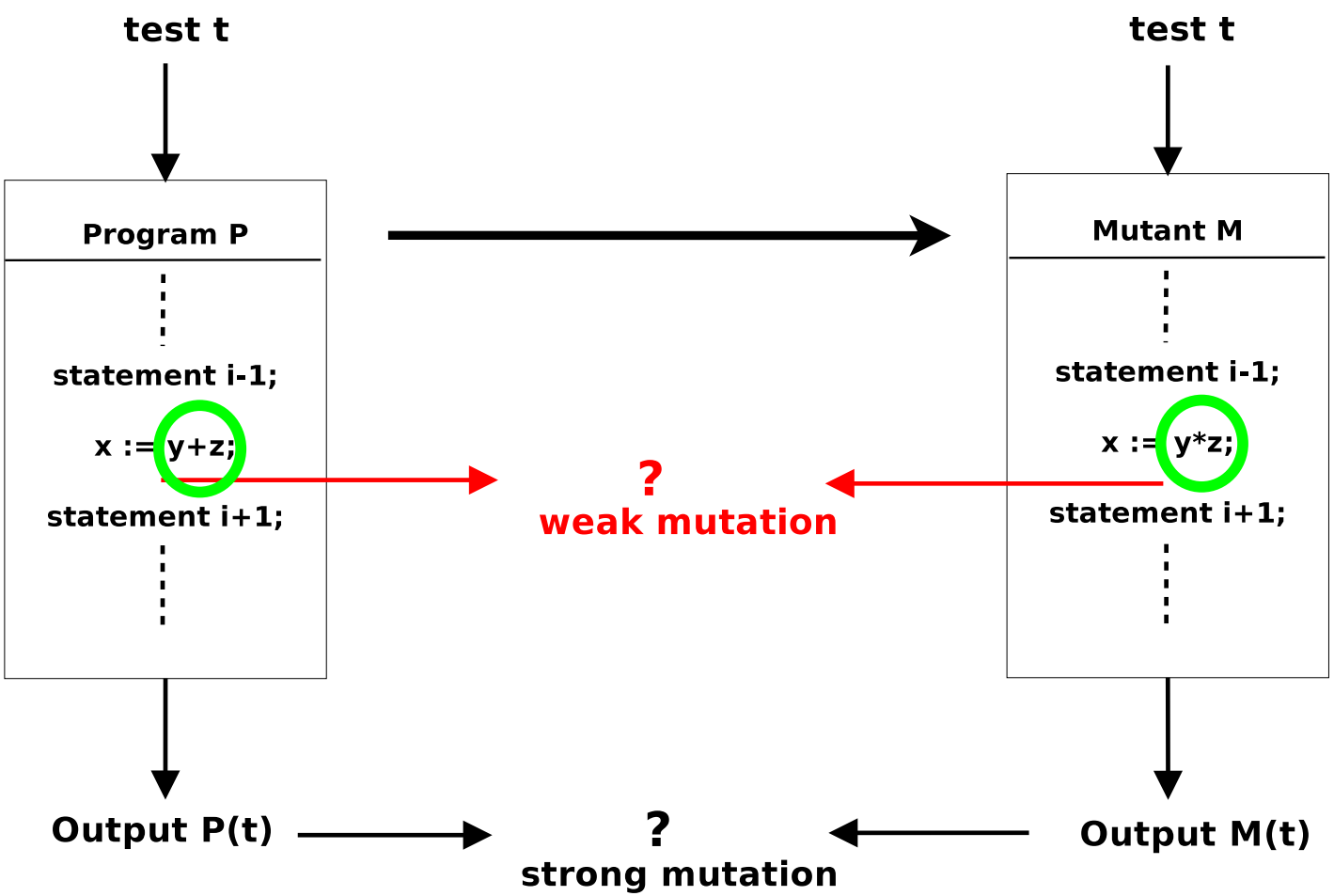
Weak mutations



Weak mutations



Weak mutations



- empirically : almost as powerful as strong mutations
- weak mutant coverage looks like predicate reachability

Outline

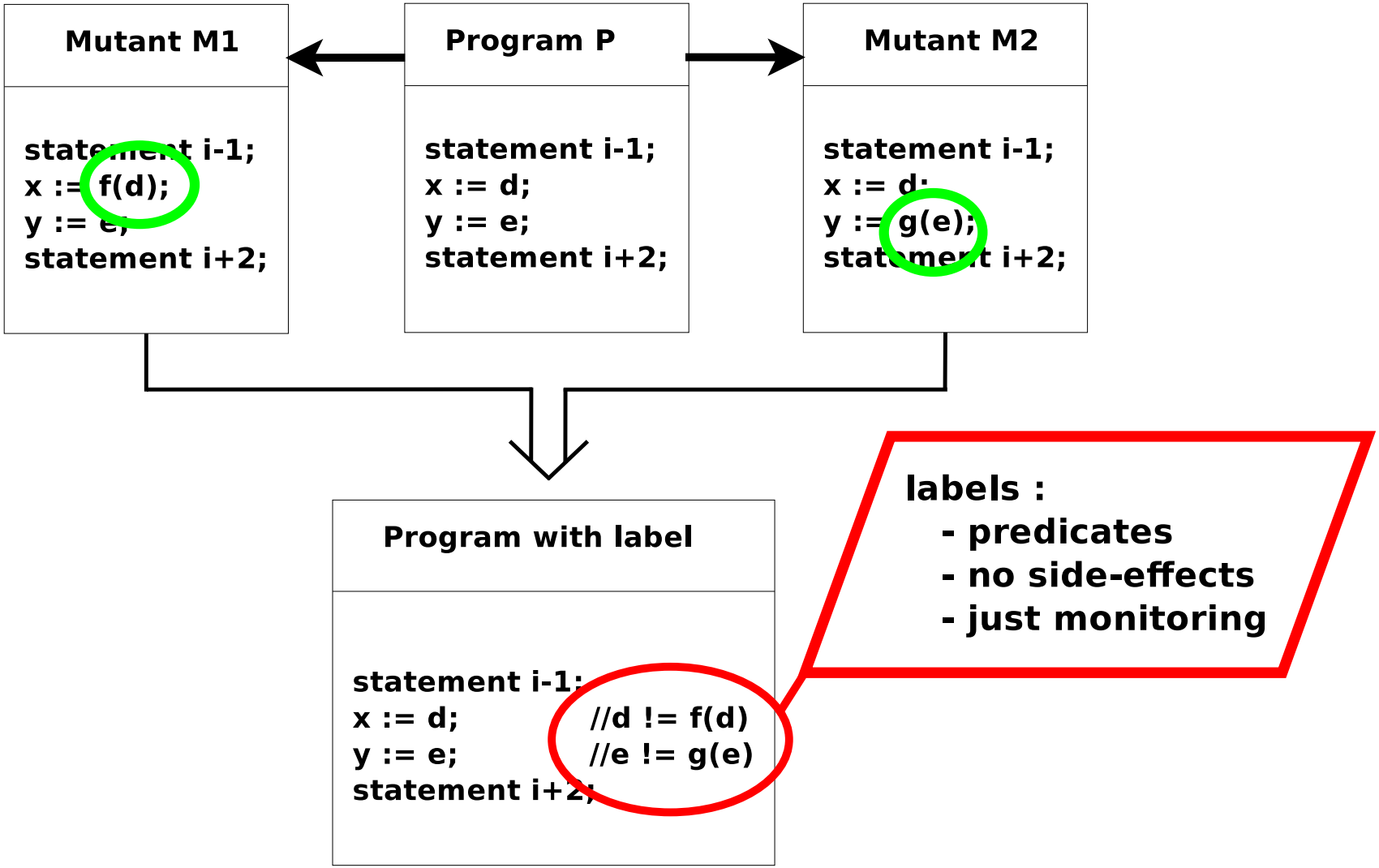
- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion

From weak mutants to labels

Push forward the (weak mutant $\leftrightarrow$ predicate reachability) analogy

Hope = reuse verification tools for mutation testing

From weak mutants to labels



From weak mutants to labels (2)

A few definitions

- label l = pair $(loc, pred)$
- program with label $= < P, L >$, with L a set of labels
- a test data t covers l : $P(t)$ reaches loc and satisfies $pred$
- $TS \models_C P$ denotes " TS fully covers criterion C on P"

Theorem

For all program P and side-effect free mutation operator op, we can deduce an annotated program $< P, L >$ such that :
for any test set TS, $TS \models_L P$ iff $TS \models_{\mathcal{M}(op)} P$

A systematic transformation

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`
- add label : $e \neq e'$

Mutation inside a decision

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

A systematic transformation (2)

We can use simpler transformations in many cases

- `x` $\mapsto$ `-x` : add label $x \neq 0$
- `x+y` $\mapsto$ `x-y` : add label $y \neq 0$
- `x >= y` $\mapsto$ `x > y` : add label $x = y$
- ...

Labels and weak mutations

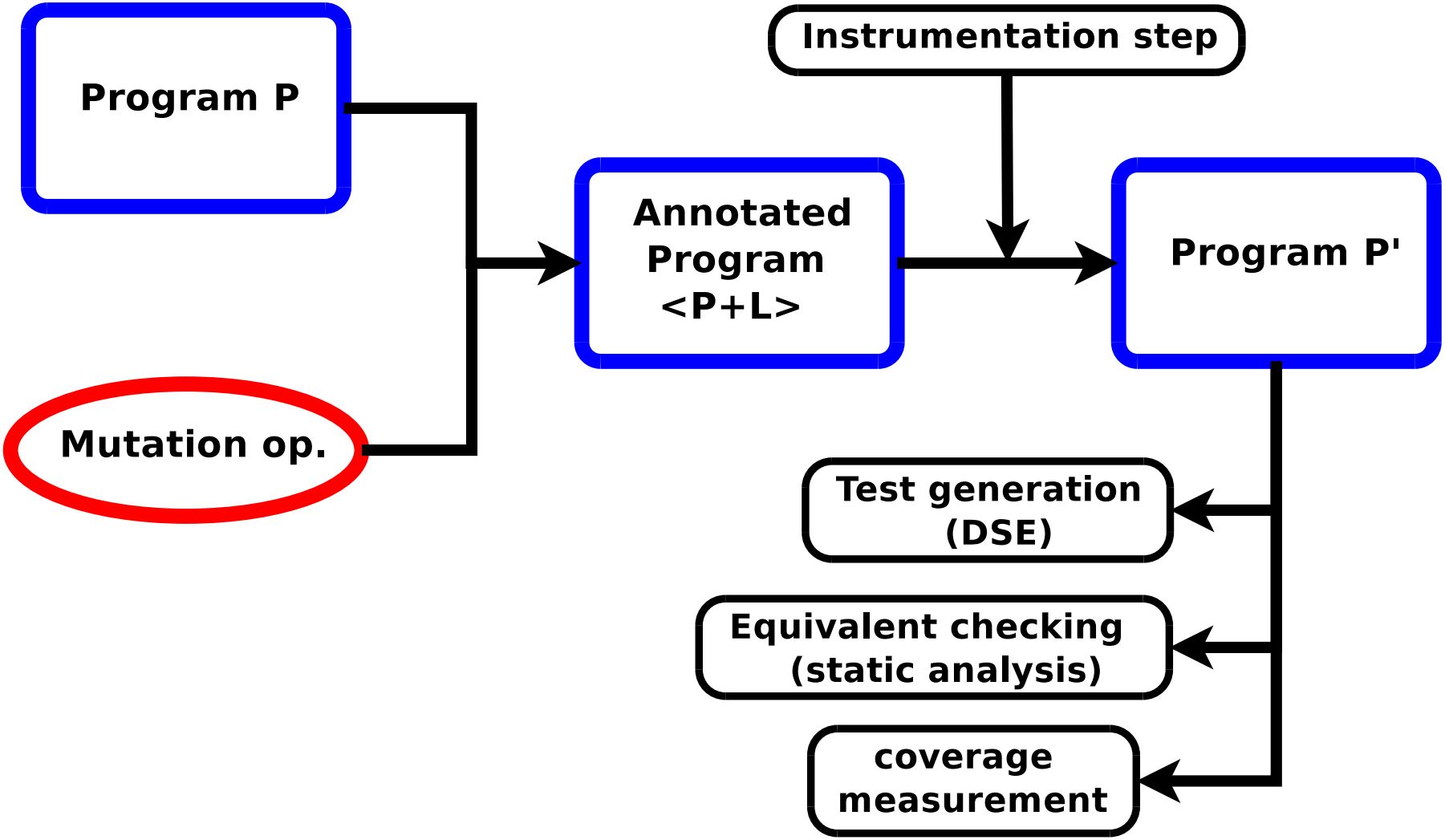
Strong analogies

Weak mutant	$\longleftrightarrow$	Label
Weakly killed (covered) mutant	$\longleftrightarrow$	Covered label
Mutation score	$\longleftrightarrow$	Label coverage rate
Equivalent mutant	$\longleftrightarrow$	Uncoverable labels

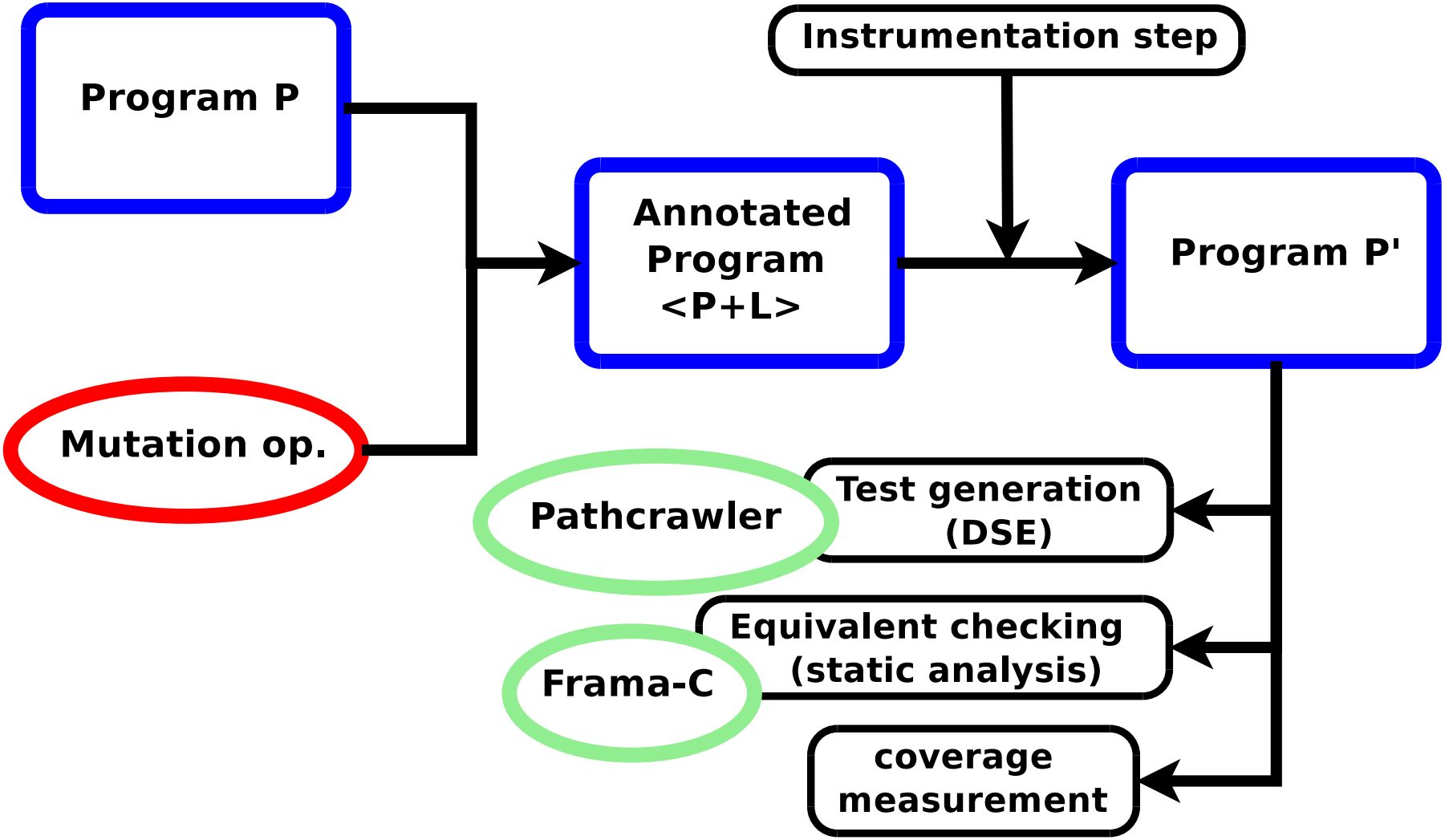
But label coverage is easier to automatize

- $\approx$ predicate reachability
- instrumentation + reuse of verification tools

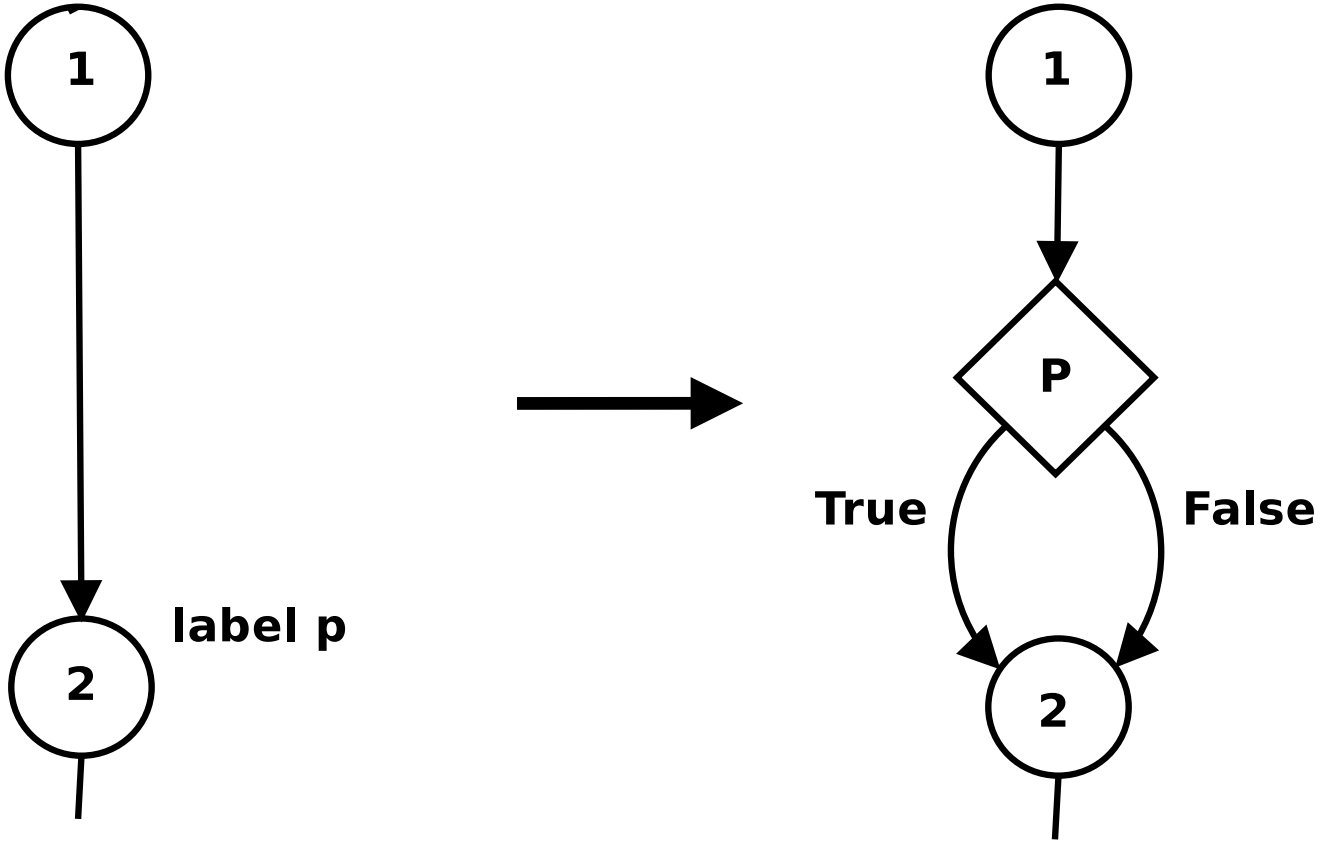
Toward automatic mutation testing



Toward automatic mutation testing



Direct instrumentation



covering label p $\equiv$ covering branch `True`

Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion

Direct instrumentation is not good enough

- Launch the PathCrawler tool (PC) on a few instrumented P'
- Compare with speed of PC launch on P

		PC on P (basic)	PC on P' (instrumentation)
Trityp	50 LOC 141 labels	0 s 14 TC	100% 466 s 63 TC
Replace	100 LOC 79 labels	2 s 121 TC	98% 1 745 s 275 TC
TCas	124 LOC 111 labels	4 s 164 TC	96% 228 767 s 249 TC

Direct instrumentation is not good enough

- Launch the PathCrawler tool (PC) on a few instrumented P'
- Compare with speed of PC launch on P

		PC on P (basic)	PC on P' (instrumentation)
Trityp	50 LOC 141 labels	0 s 14 TC	100% 466 s 63 TC
Replace	100 LOC 79 labels	2 s 121 TC	98% 1 745 s 275 TC
TCas	124 LOC 111 labels	4 s 164 TC	96% 228 767 s 249 TC

✓ Achieve high mutation coverage

× Very significant overhead ! ! !

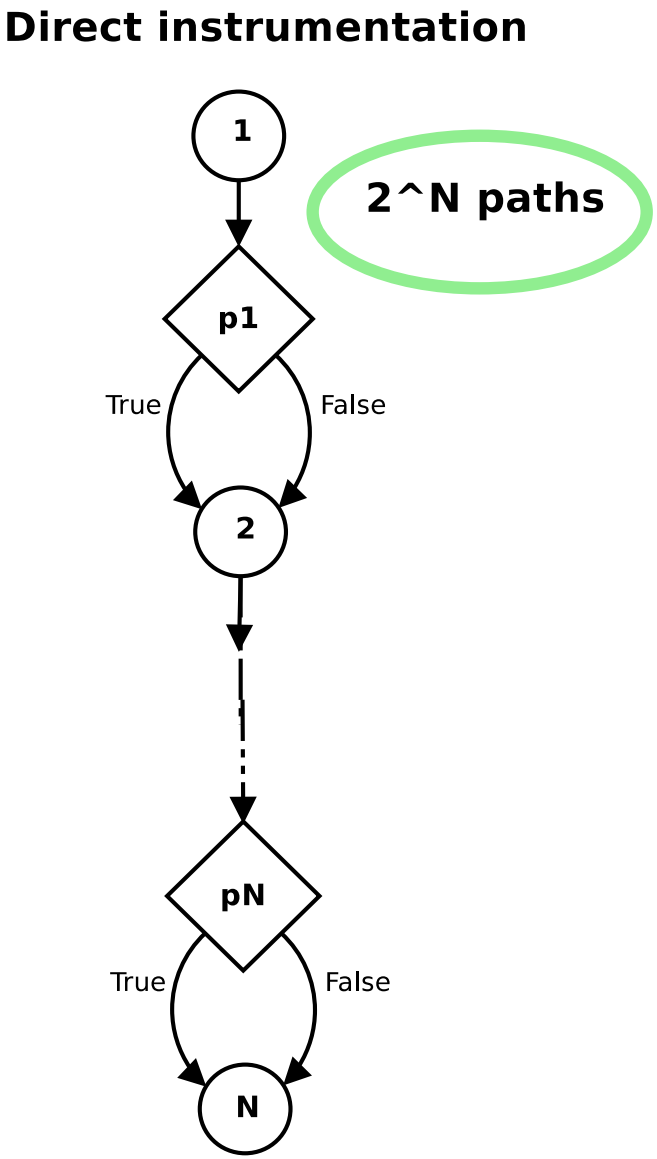
Direct instrumentation is not good enough (2)

P' has exponentially more paths than P

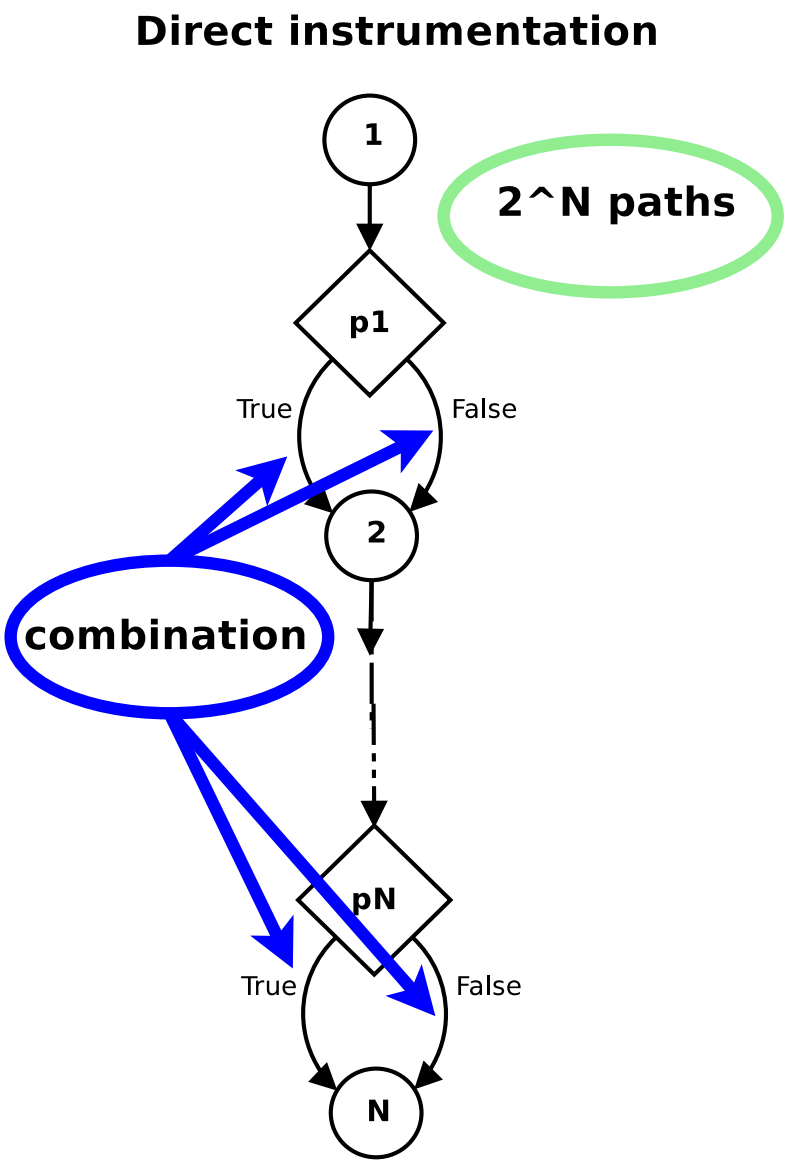
Paths in P' are much more more complex than needed

- at each "label", an execution σ is required to cover p or $\neg p$
- an execution $\sigma_1\sigma_2\ldots\sigma_n$ in P can become $\sigma_1(l_1)\sigma_2(\neg l_2)\ldots(l_n)\sigma_n$ in P'

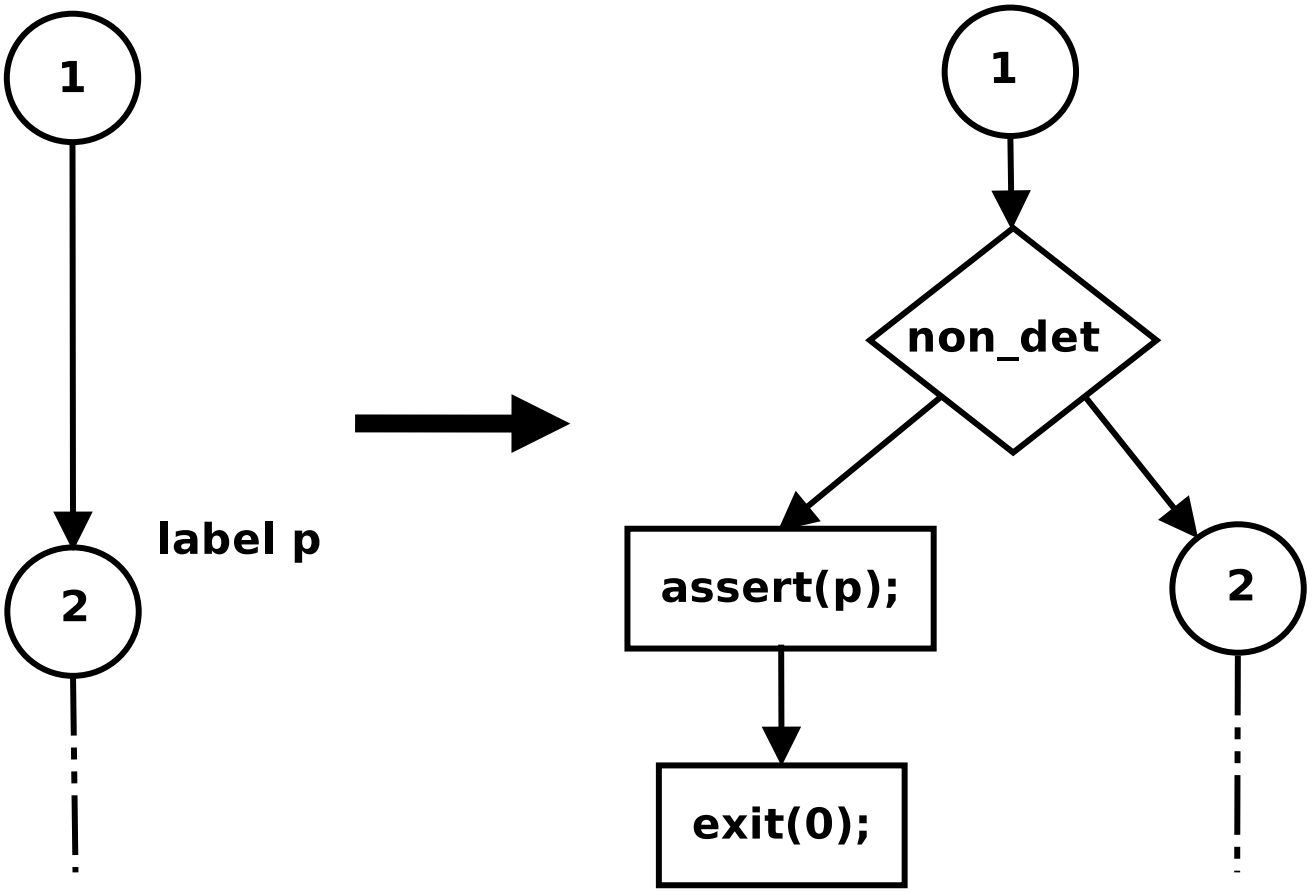
Direct instrumentation is not good enough (2)



Direct instrumentation is not good enough (2)



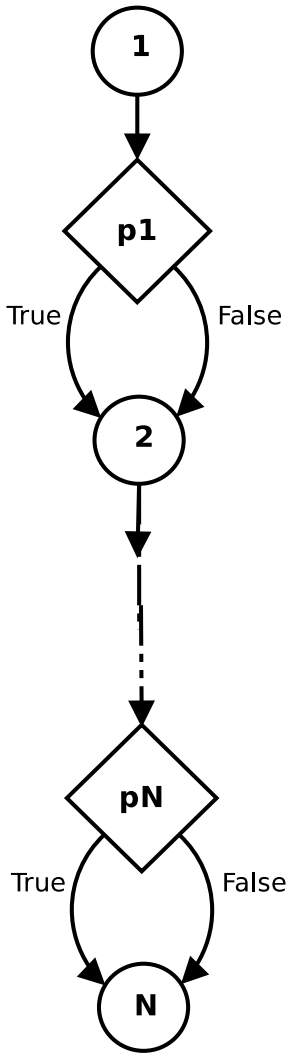
Optim 1 : smarter instrumentation



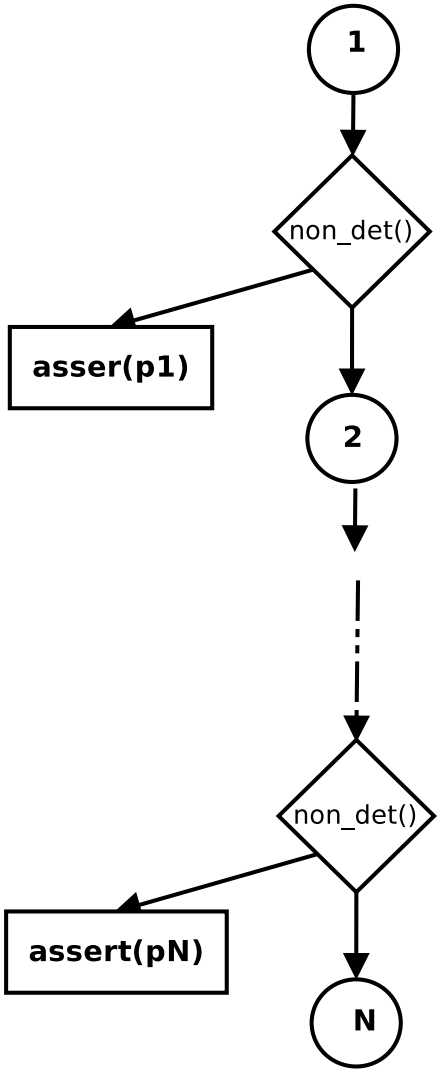
- ✓ Paths in P" have at most one "label constraint"
- ✓ P" has exponentially less paths than P'

Optim 1 : smarter instrumentation

Direct instrumentation

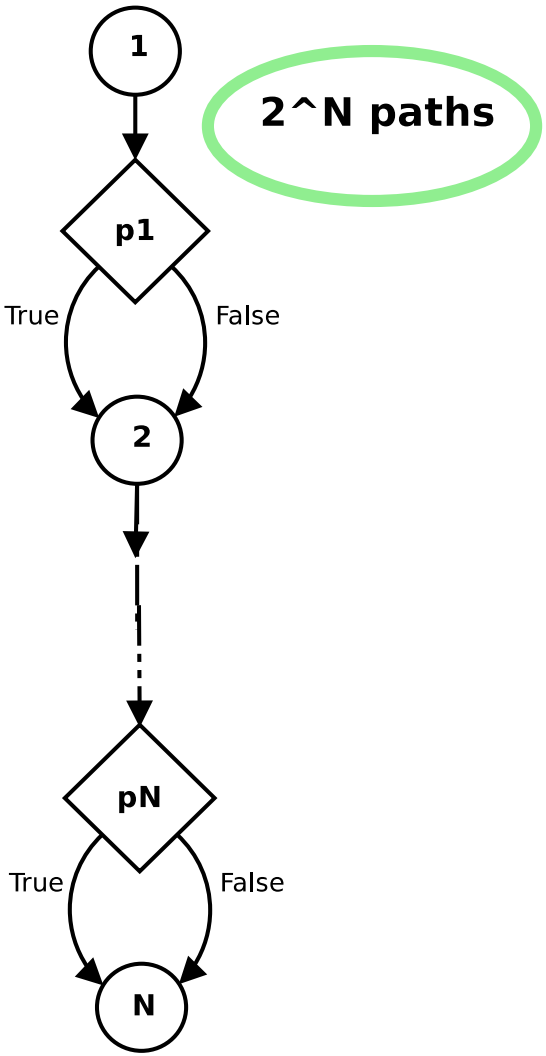


Smarter Instrumentation

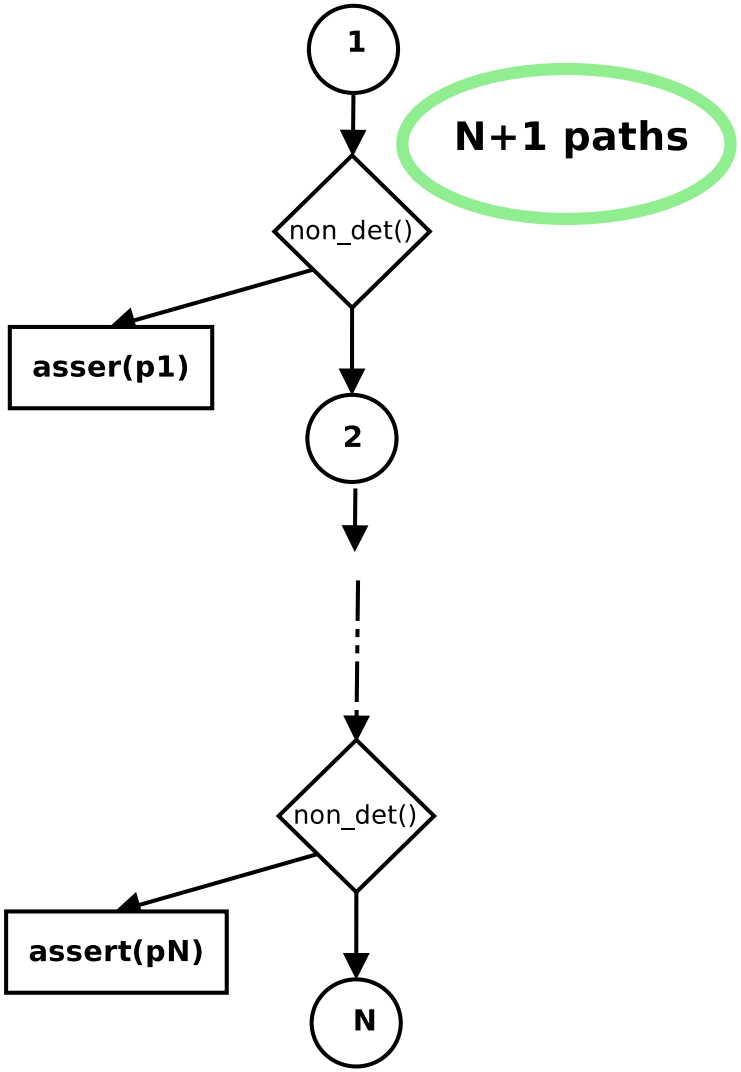


Optim 1 : smarter instrumentation

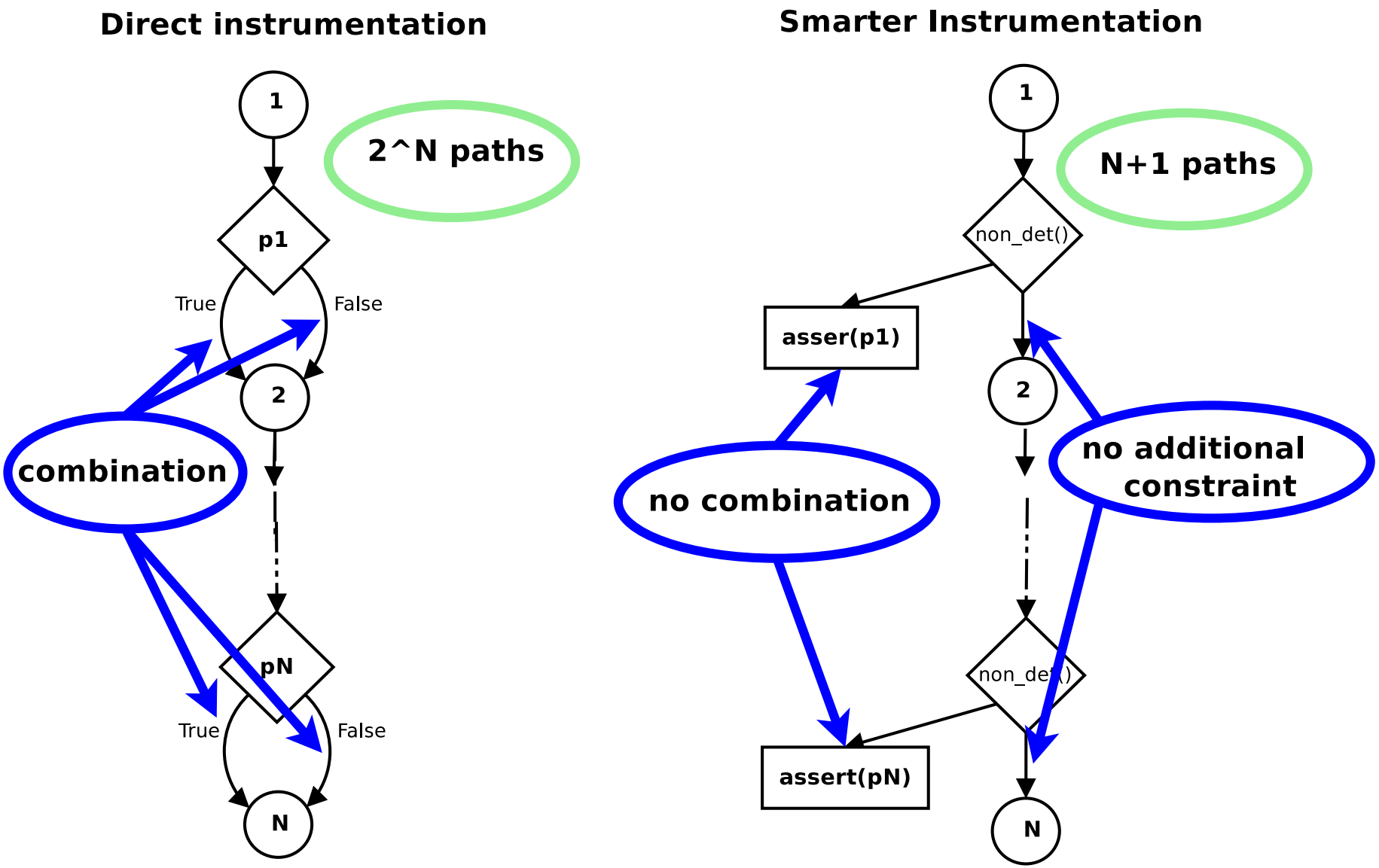
Direct instrumentation



Smarter Instrumentation



Optim 1 : smarter instrumentation



Optim 2 : incremental DSE

Observations

- we need to cover each label only once
- yet, already covered labels "stay in place" in P'
  - burden the DSE tool with "useless" paths

Optim 2 : incremental DSE

Observations

- we need to cover each label only once
- yet, already covered labels "stay in place" in P'
  - burden the DSE tool with "useless" paths

Our approach : incremental DSE

- Partition labels into disjoint sets L_1, ..., L_k
- Choose a set L_k
- Launch PathCrawler on $< P, L_k >$
- when terminates :
  - get the generated test set TS
  - prune from all L_i the labels covered by TS
- loop forall L_i

Results

		PC on P (basic)	PC on P' (direct)	PC on P'' (smart)
Trityp	50 LOC 141 labels	0 s 14 TC	100% 466 s 63 TC	100% 1 s 84 TC
Replace	100 LOC 79 labels	2 s 121 TC	98% 1 745 s 275 TC	100% 50 s 393 TC
TCas	124 LOC 111 labels	4 s 164 TC	96% 228 767 s 249 TC	100% 72 s 1 049 TC

- ✓ higher mutation coverage
- ✓ significant speedup wrt. direct instrumentation
- yet, significant overhead (20x) wrt. uninstrumented version

Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion

Coverage computation

- use Emma (decision coverage) and MuJava (mutation score)
- 100 tests

		Emma		MuJava
		P (witness)	P'	P + M
TCas	124 LOC 111 labels	0,03 s	0,03 s	5 s
Replace	436 LOC 607 labels	0,05 s	0,10 s	40 s
Jtopas	5400 LOC 610 labels	3,22 s	11,72 s	1 400 s

Results

- significant speed-up (x40 - x100)
- very reasonable overhead (x1 - x4)

Detection of uncoverable labels ($\approx$ equiv. mutants)

We use the Frama-C framework to detect uncoverable labels

- Value Analysis plugin (VA)
- restricted logical reasoning (L) [ad hoc plugin]

	# labels		Categories of uncovered			
	total	uncovered	VA	L	M	Unknown
Trityp	141	6	4	0	2	0
TCas	111	13	6	3	4	0
Replace	79	12	11	1	0	0

VA (resp. L) : # labels proved uncoverable through VA (resp. L)

M : # labels manually proved uncoverable

Results [direct instrumentation]

- good detection rate (0.6 - 1) with generic-purpose analysis
- can probably be improved

Outline

- Introduction
- Mutation testing
- From mutants to programs with labels
- DSE and mutants
- Other results
- Conclusion

Conclusion & future work

A subtantial subset of weak mutations can be reduced to predicate reachability

Allow to reuse the whole (symbolic) "verification machinery"

We have shown first encouraging expriments for

- automatic test generation with high mutation score
- automatic discovery of equivalent mutants
- efficient coverage measurement

Future work

- more experiments
- native support for labels in PathCrawler (goal = low overhead)
- dedicated mutant equivalence checking plugin in Frama-C ?