Symbolic Execution and Advanced Test Coverage Criteria

Nikolai Kosmatov

joint work with Sébastien Bardin, Omar Chebaro, Mickaël Delahaye. . .

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

USE 2015, Oslo, June 23, 2015

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

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)

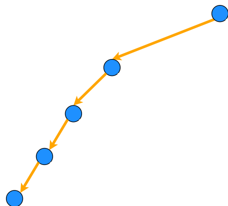
input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- 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 $Paths^{\leq k}(\mathtt{P})$

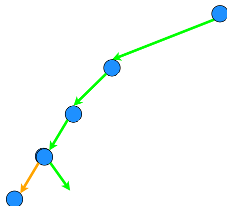
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $< \text{s}, \sigma >$ into *TS*
- loop until no more paths to cover



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

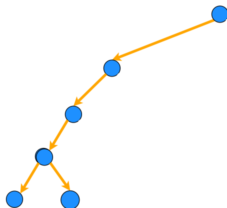


Dynamic Symbolic Execution (2)

input : a program P

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

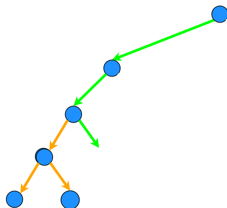
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $< \text{s}, \sigma >$ into *TS*
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

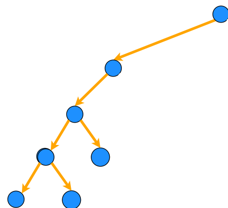
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $<$ s, $\sigma >$ into *TS*
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

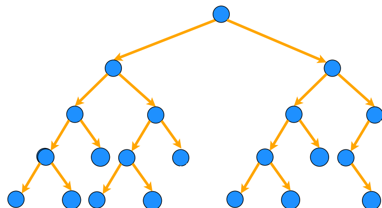
- pick an uncovered path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $<$ s, $\sigma >$ into TS
- loop until no more paths to cover



Dynamic Symbolic Execution (2)

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

- pick an uncovered path $\sigma \in \mathit{Paths}^{\leq k}(\texttt{P})$
- is the path predicate φ_σ satisfiable ? [smt solver]
- if SAT(s) then add a new pair $< \mathrm{s}, \sigma >$ 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

Our proposal

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 a black-box manner

Implem. in PathCrawler

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

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

Outline

- Introduction
- Simulation of coverage criteria by labels
- Efficient DSE for labels
- Experiments
- Conclusion

Labels and the notion of simulation

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

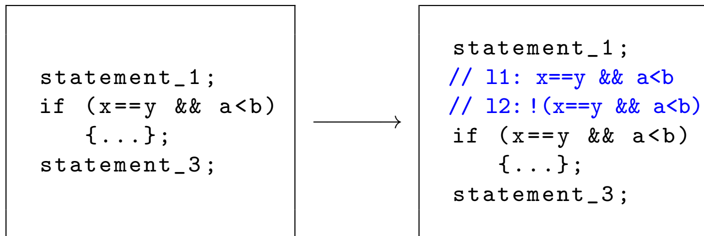
- φ is a well-defined predicate in P at location *loc*
- φ contains no side-effect expression

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**.

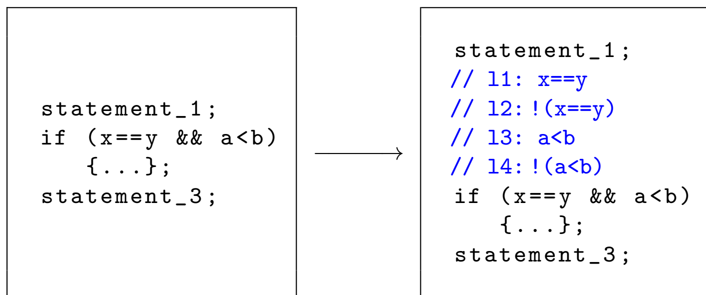
Goal : show the relative expressiveness of **LC**

Simulation of coverage criteria by labels : DC



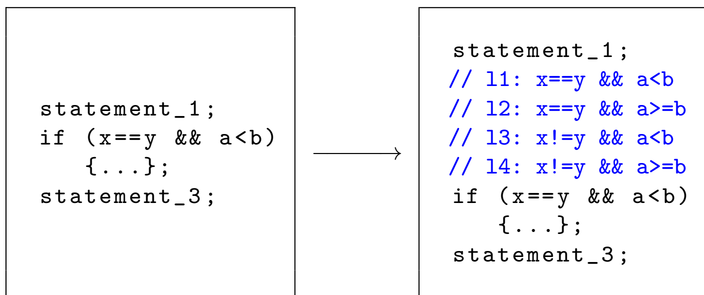
Decision Coverage (**DC**)

Simulation of coverage criteria by labels : CC



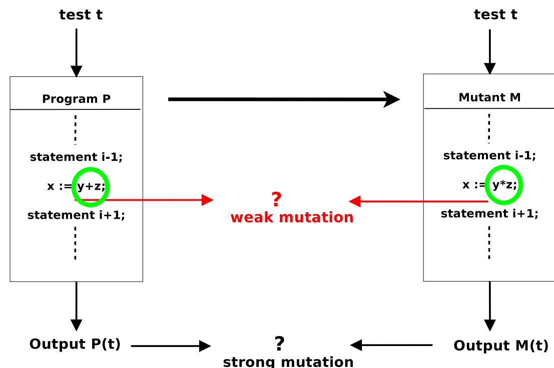
Condition Coverage (**CC**)

Simulation of coverage criteria by labels : MCC



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**.

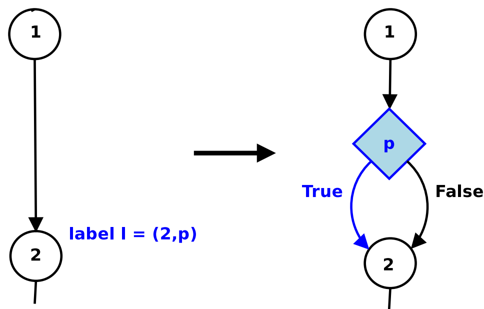
Goals

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

Outline

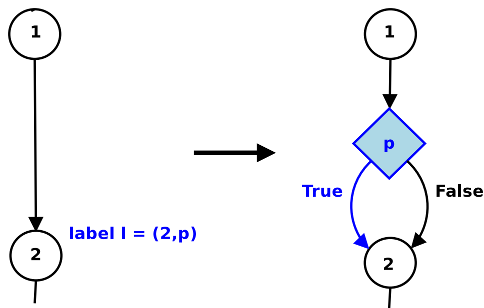
- Introduction
- Simulation of coverage criteria by labels
- Efficient DSE for labels
- Experiments
- Conclusion

Direct instrumentation P' [APEX, Mutation DSE]



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

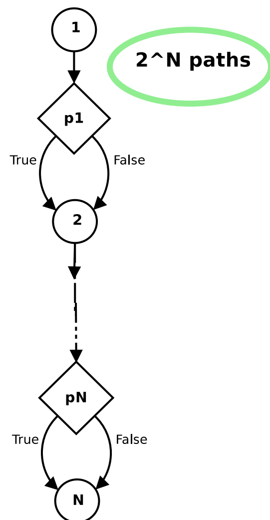
Direct instrumentation P' [APEX, Mutation DSE]



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

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

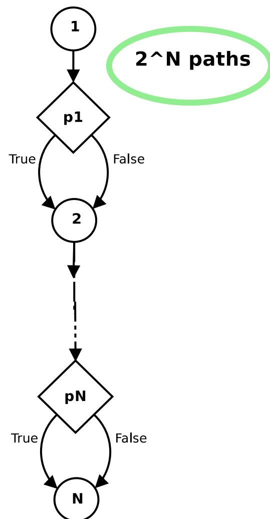
Direct instrumentation P' is not good enough



Direct instrumentation P' is not good enough

Non-tightness 1

- ✗ P' has exponentially more paths than P



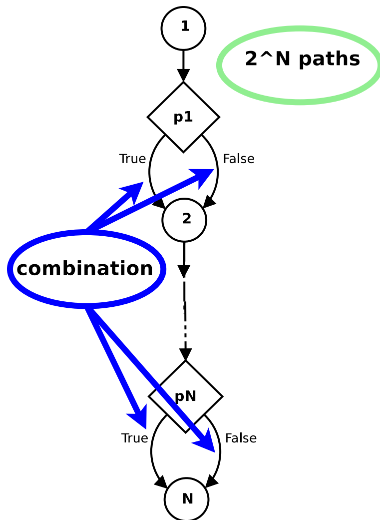
Direct instrumentation P' is not good enough

Non-tightness 1

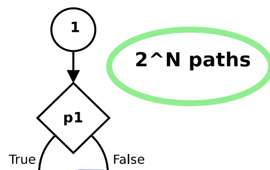
- ✗ P' has exponentially more paths than P

Non-tightness 2

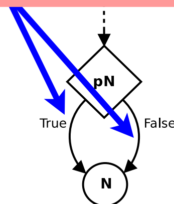
- ✗ Paths in P' too complex
  - at each label, require to cover p or to cover $\neg p$
  - π' covers up to N labels



Direct instrumentation P' is not good enough



- ✓ sound & complete instrumentation w.r.t. **LC**
- ✗ dramatic overhead [theory & practice]

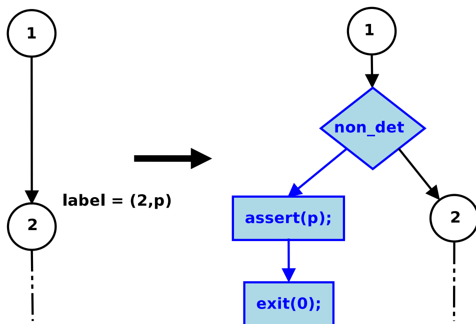


Our approach

The DSE* algorithm

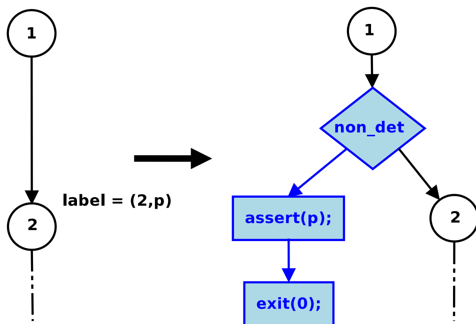
- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in a black-box manner

DSE* : Tight Instrumentation $P^\star$



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

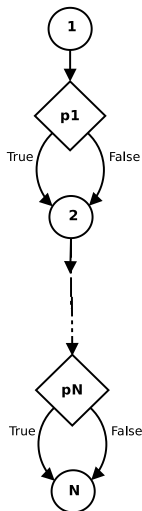
DSE* : Tight Instrumentation $P^\star$

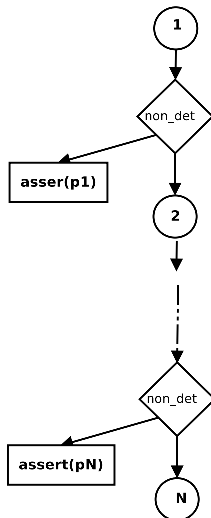


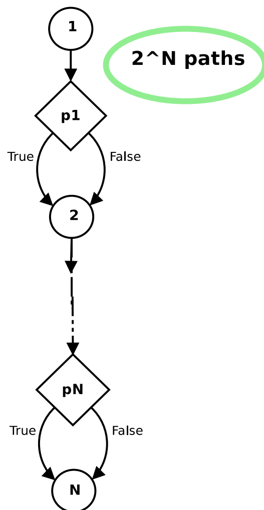
Covering label l $\Leftrightarrow$ Covering `exit(0)`

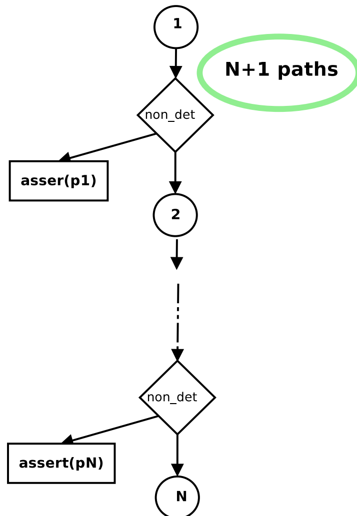
✓ sound & complete instrumentation w.r.t. **LC**

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

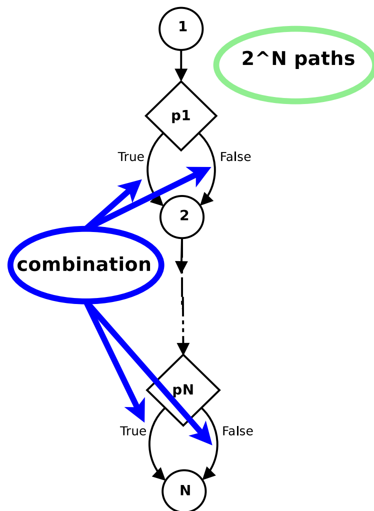





Direct instrumentation
1
2^N paths
p1
True
False
2
pN
True
False
N


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

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



Tight Instrumentation

1

N+1 paths

non_det

asser(p1)

2

no combination

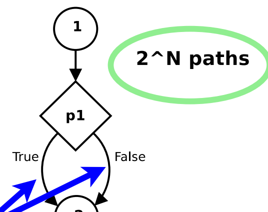
no additional constraint

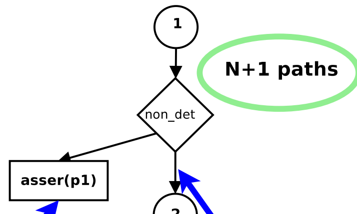
non_det

assert(pN)

N

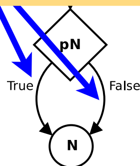
DSE* : Direct vs tight instrumentation, P' vs $P^\star$

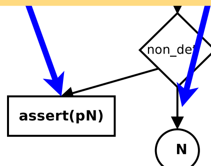


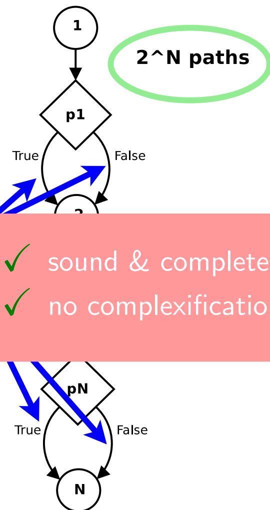


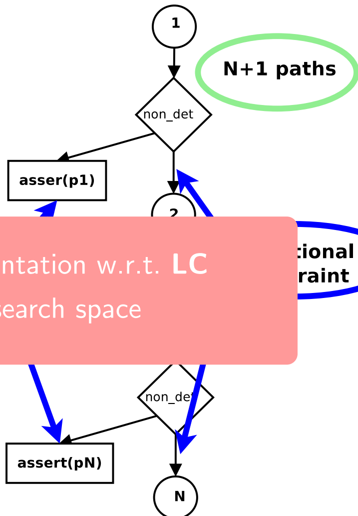
Tightness

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









- ✓ 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 $P^\star$ ending in already-covered labels
- we burden DSE with "useless" paths w.r.t. **LC**

Solution : Iterative Label Deletion

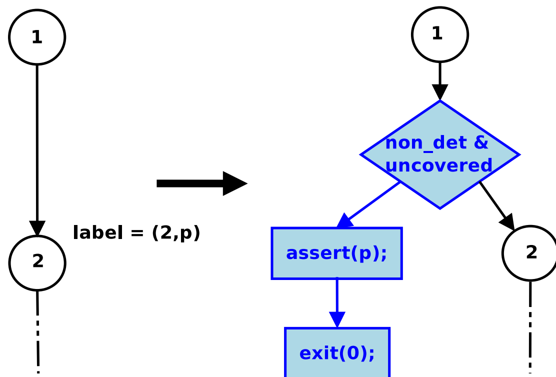
- keep a *covered/uncovered* status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates the status [truly requires DSE]

Implementation

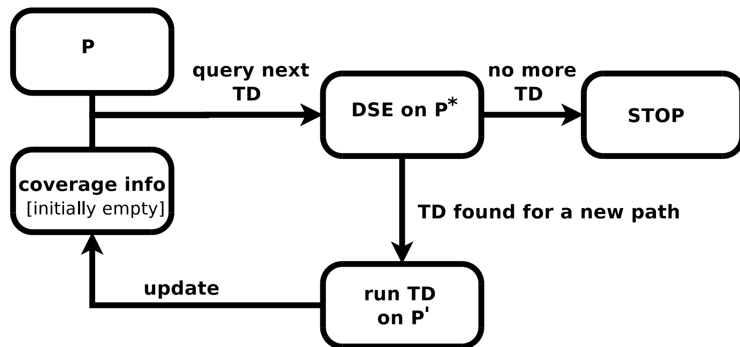
- symbolic part : a slight modification of $P^\star$
- dynamic part : a slight modification of P'

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

- Introduction
- Simulation of coverage criteria by 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 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%]

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%]

Conclusion

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

A few detailed results

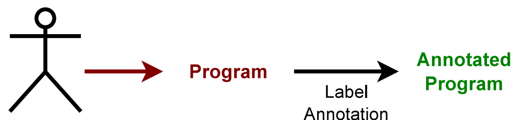
			DSE	DSE'	DSE*
utf8-5	wm	#paths	680	**11,111**	**743**
108 loc	84 *l*	time	2s	40s	8.1s
		cover		82/84	82/84
utf8-7	wm	#paths	3,069	**81,133**	**3,265**
108 loc	84 *l*	time	5.8s	**576s**	**35s**
		cover		82/84	82/84
tcas	wm	#paths	4,420	**300,213**	**6,014**
124 loc	111 *l*	time	5.6s	**662s**	**27s**
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

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

The LTEST toolset for labels [TAP 14]



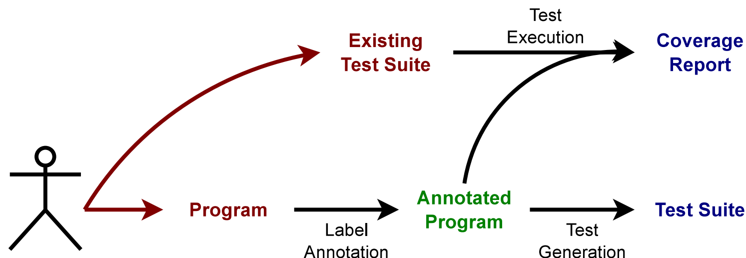
Supported criteria

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

Encoded with labels [ICST 2014]

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

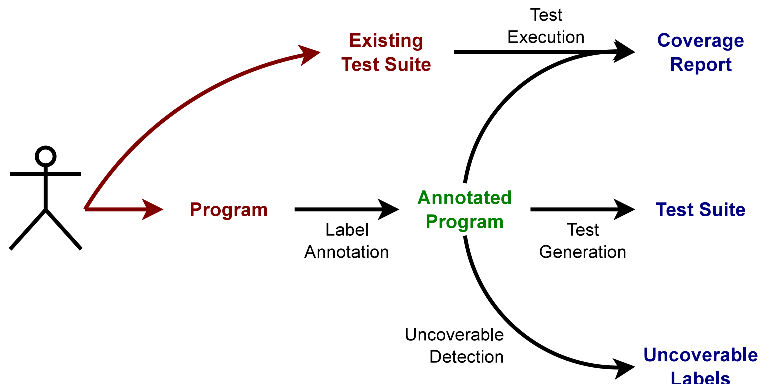
The LTest toolset for labels [TAP 14]



DSE* procedure [ICST 2014]

- DSE with native support for labels
- extension of PathCrawler

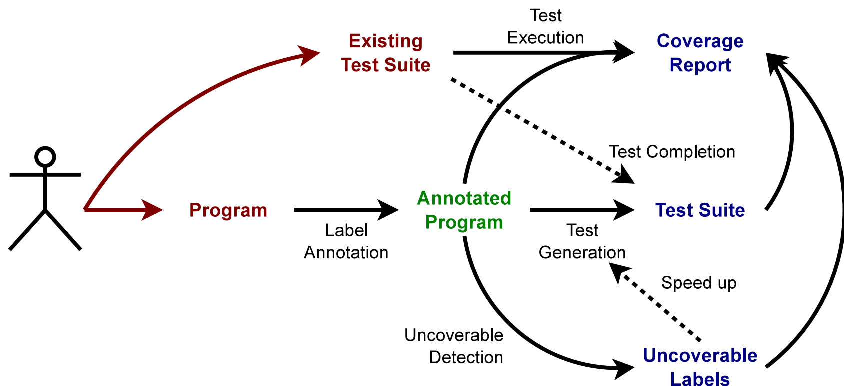
The LTEST toolset for labels [TAP 14]



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`, `?`

Outline

- Introduction
- Simulation of coverage criteria by 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