

Enhancing Symbolic Execution for Coverage-Oriented Testing

Sébastien Bardin, Nikolai Kosmatov, Mickaël Delahaye

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

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

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

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 P on input v , and a given path σ
- a **path predicate** φ_σ for σ is a formula s.t.
$$v \models \varphi_\sigma \Rightarrow P(v) \text{ follows } \sigma$$
- can be used for bounded-path testing!
- old idea, recent renew interest [requires powerful solvers]

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 P on input v , and a given path σ

- a **path predicate** φ_σ for σ is a formula s.t.

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

- 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]

input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]

Context : Dynamic Symbolic Execution (2)

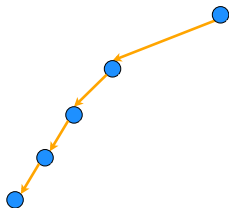
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



Context : Dynamic Symbolic Execution (2)

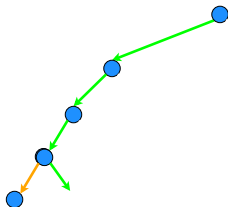
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



Context : Dynamic Symbolic Execution (2)

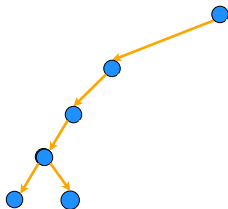
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



Context : Dynamic Symbolic Execution (2)

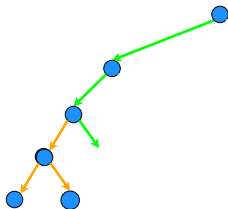
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



Context : Dynamic Symbolic Execution (2)

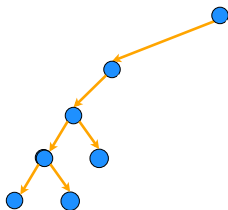
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



Context : Dynamic Symbolic Execution (2)

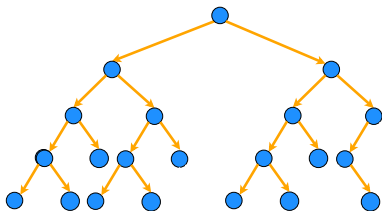
input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $\langle s, \sigma \rangle$
- loop until no more path to cover

[wpre, spost]

[smt solver]



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
-

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.]

Recent efforts [Active Testing, Augmented DSE, Mutation DSE]

- limited or unclear expressiveness
- explosion of the search space [APEX : 272x avg, up to 2,000x]

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
-

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

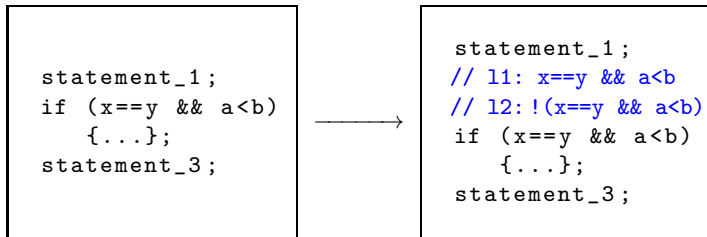
- support these criteria in a **unified way**
- support these criteria in an **efficient way**
- detect (some) infeasible test requirements

Results

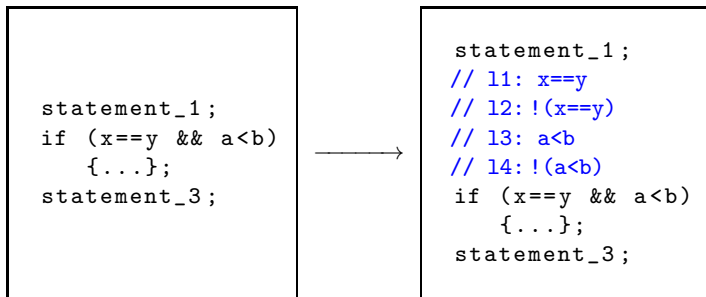
- ✓ generic low-level encoding of coverage criteria [ICST 14]
- ✓ efficient variant of DSE for coverage criteria [ICST 14]
- ✓ sound and quasi-complete detection of infeasibility [ICST 15]

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

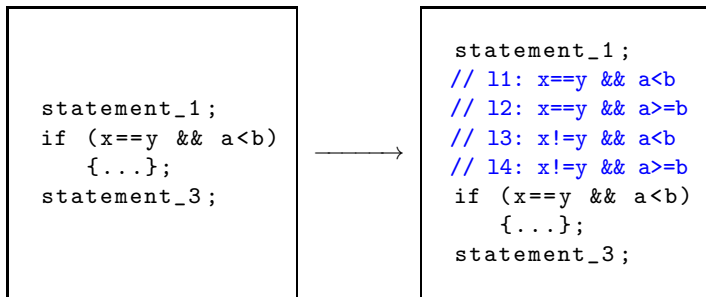
- Annotate programs with **labels**
 - ▶ predicate attached to a specific program instruction
- Label (loc, φ) is covered if a test execution
 - ▶ reaches the instruction at loc
 - ▶ satisfies the predicate φ
- **Good for us**
 - ▶ can easily encode a large class of coverage criteria [see after]
 - ▶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



Decision Coverage (**DC**)



Condition Coverage (CC)



Multiple-Condition Coverage (**MCC**)

OBJ : generic specification mechanism for coverage criteria ✓

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

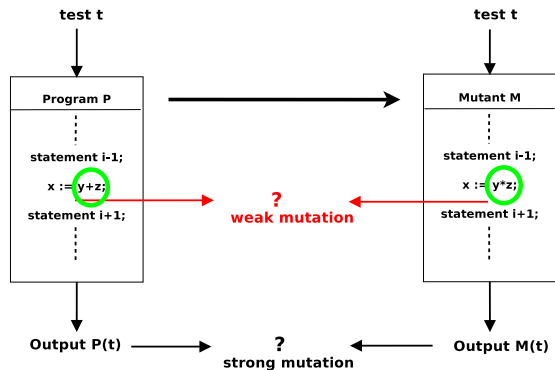
OBJ : generic specification mechanism for coverage criteria ✓

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

Out of scope :

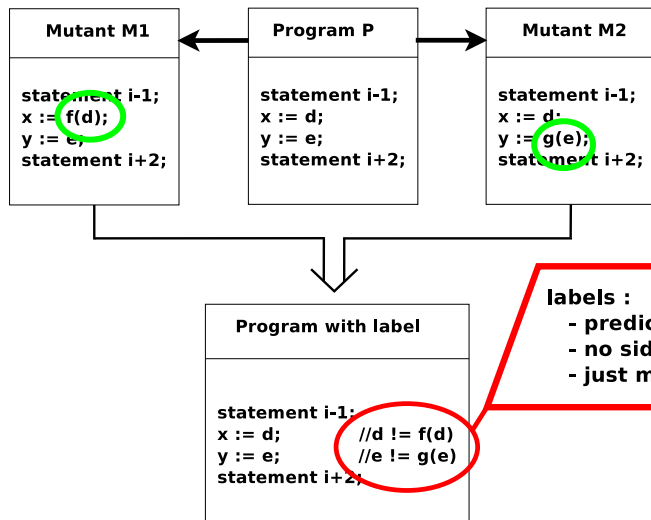
- . strong mutations, MCDC
- . (side-effect weak mutations)

Focus : Simulation of Weak Mutations



- 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 (1)



One label per mutant

Mutation inside a statement

- $lhs := e \quad \mapsto \quad lhs := e'$
 - ▶ add label : $e \neq e'$
- $lhs := e \quad \mapsto \quad lhs' := e$
 - ▶ add label : $\&lhs \neq \&lhs' \wedge (lhs \neq e \vee lhs' \neq e)$

Mutation inside a decision

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

Beware : no side-effect inside labels

One label per mutant

Mutation inside a statement

■ $lhs := e \quad \mapsto \quad lhs := e'$

Theorem

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

Mutation inside a decision

■ $if (cond) \quad \mapsto \quad if (cond')$

▶ add label : $cond \oplus cond'$

Beware : no side-effect inside labels

One label per mutant

Mutation inside a statement

■ $lhs := e \quad \mapsto \quad lhs := e'$

Theorem

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

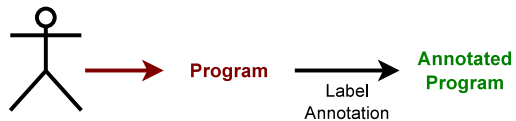
Mutation inside a decision

■ $if (cond) \quad \mapsto \quad if (cond')$

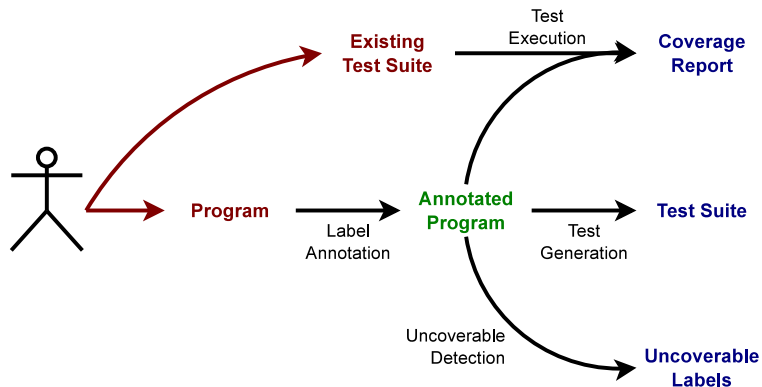
▶ add label : $cond \oplus cond'$

Beware : no side-effect inside labels

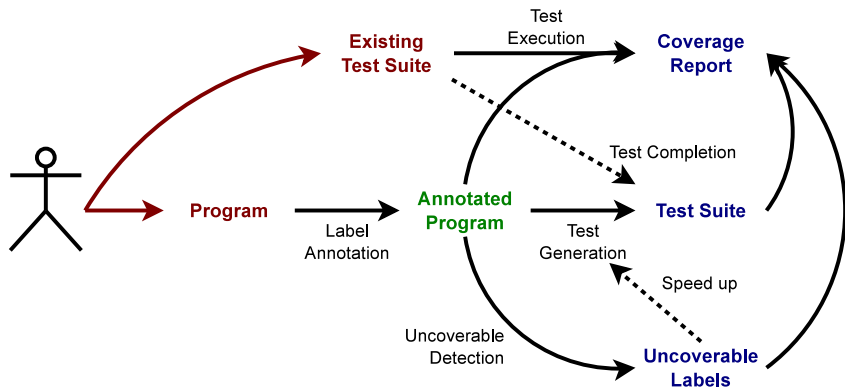
- plugin of the FRAMA-C analyser for C programs
 - ▶ open-source
 - ▶ sound, industrial strength
 - ▶ among other : VA, WP, specification language
- based on PATHCRAWLER for test generation



LTEST overview

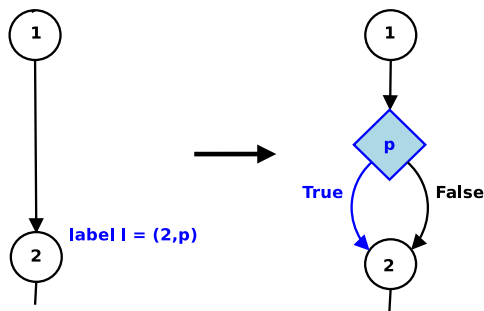


LTEST overview



- 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
- ✗ complexification of the search space [#paths, shape of paths]
- ✗ dramatic overhead [theory & practice] [Apex : avg 272x, max 2000x]

Direct instrumentation is not good enough

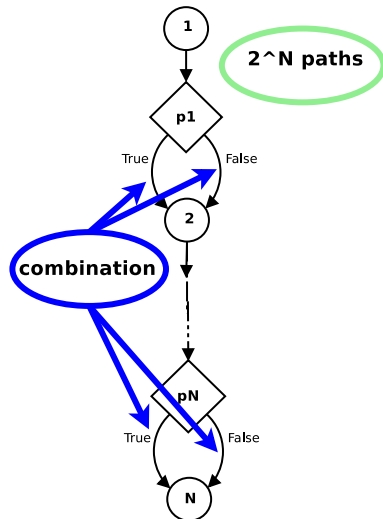
Non-tightness 1

- ✗ 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

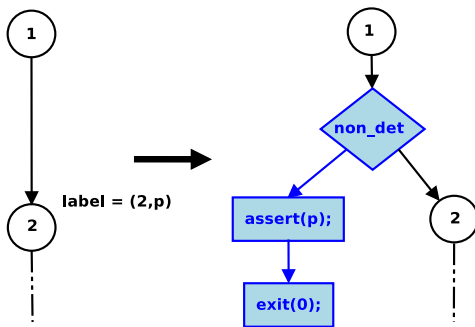
Direct instrumentation



The DSE* algorithm [ICST 14]

- Tight instrumentation P^* : totally prevents “complexification”
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in black-box

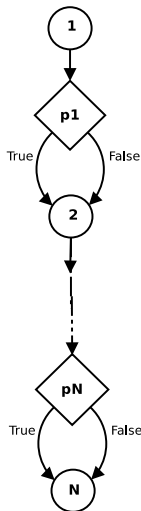
DSE* : Tight Instrumentation



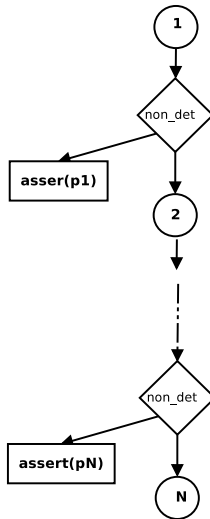
Covering label l \Leftrightarrow Covering `exit(0)`

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

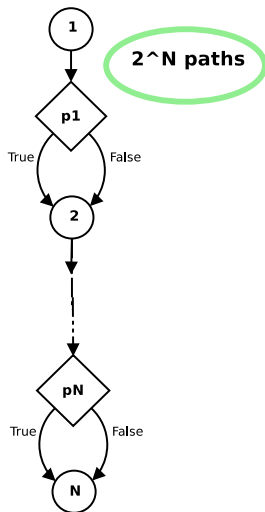
Direct instrumentation



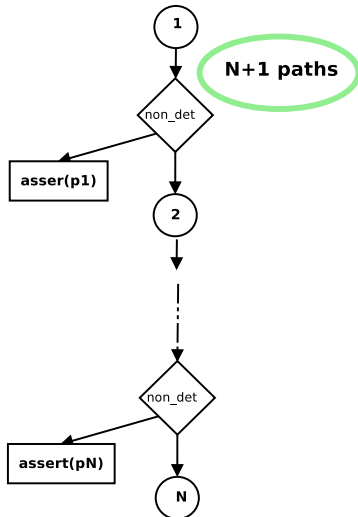
Tight Instrumentation



Direct instrumentation

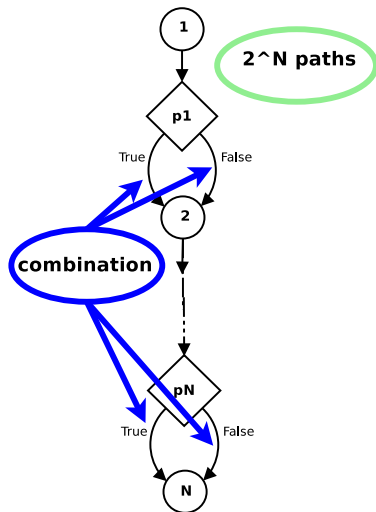


Tight Instrumentation

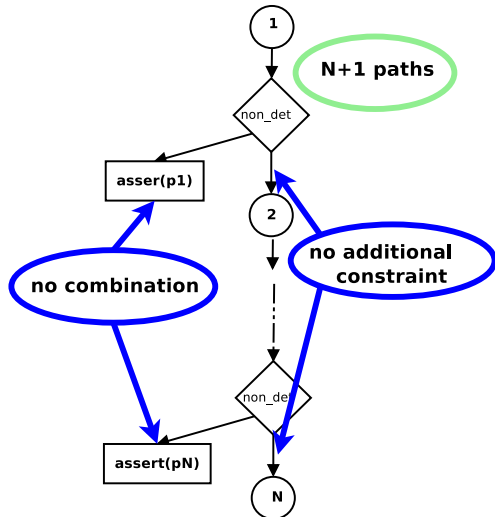


DSE* : Tight Instrumentation (2)

Direct instrumentation



Tight Instrumentation



Observations

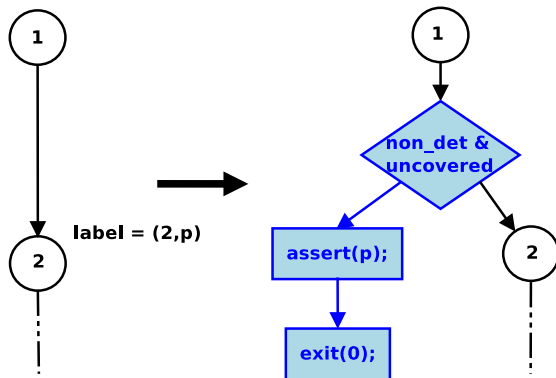
- we need to cover each label only once
- yet, DSE explores paths of P^* ending in already-covered labels
- burden DSE with “useless” paths w.r.t. label coverage

Solution : Iterative Label Deletion

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

Iterative Label Deletion is relatively complete w.r.t. label coverage

DSE* : Iterative Label Deletion (2)



The DSE* algorithm

- Tight instrumentation P^* : totally prevents “complexification”
- Iterative Label Deletion : discards some redundant paths relative completeness w.r.t. label coverage
- Both techniques can be implemented in black-box

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]

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

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

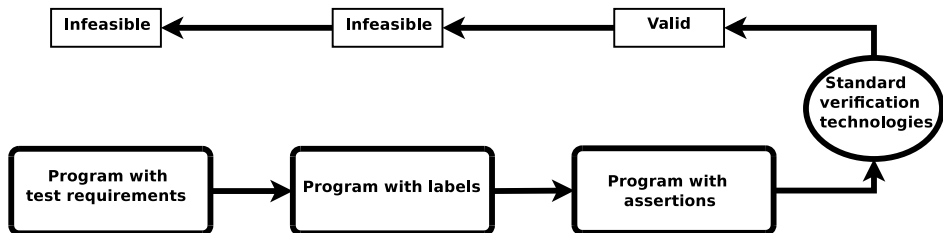
- Introduction
- Labels
- Efficient DSE for Labels
- Infeasible label detection
- The GACC criterion
- Conclusion

Basic ideas

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

Overview of the approach

- labels as a unifying criteria
- label infeasibility \Leftrightarrow assertion validity
- s-o-t-a verification for assertion checking



- only soundness is required (verif)
- ▶ label encoding **not required to be perfect**
 - . mdc and strong mutation ok

Two broad categories of sound assertion checkers

- State-approximation computation [forward abstract interp., cegar]
 - ▶ compute an invariant of the program
 - ▶ then, analyze all assertions (labels) in one go
- Goal-oriented checking [$\text{pre}^{\leq k}$, weakest precondition., cegar]
 - ▶ perform a dedicated check for each assertion
 - ▶ a single check usually easier, but many of them

Two broad categories of sound assertion checkers

- State-approximation computation [forward abstract interp., cegar]
 - ▶ compute an invariant of the program
 - ▶ then, analyze all assertions (labels) in one go
- Goal-oriented checking [$\text{pre}^{\leq k}$, weakest precondition., cegar]
 - ▶ perform a dedicated check for each assertion
 - ▶ a single check usually easier, but many of them

Focus on Value-analysis (VA) and Weakest Precondition (WP)

- correspond to our implementation
- well-established approaches
- [the paper is more generic]

Focus : checking assertion validity (2)

	VA	WP
sound for assert validity	✓	✓
blackbox reuse	✓	✓
local precision	✗	✓
calling context	✓	✗
calls / loop effects	✓	✗
global precision	✗	✗
scalability wrt. #labels	✓	✓
scalability wrt. code size	✗	✓

hypothesis : VA is interprocedural

VA and WP may 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;
    else
        res = 0;
    //l1: res == 0
}
```

VA and WP may 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;
    else
        res = 0;
    //@assert res != 0
}
```

VA and WP may 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;
    else
        res = 0;
    //@assert res != 0      // both VA and WP fail
}
```

Goal = get the best of the two worlds

- idea : VA passes to WP the global info. it lacks

Which information, and how to transfer it ?

- VA computes (internally) some form of invariants
- WP naturally takes into account assumptions `//@ assume`

solution VA exports its invariants on the form of WP-assumptions

Goal = get the best of the two worlds

- idea : VA passes to WP the global info. it lacks

Which information, and how to transfer it ?

- VA computes (internally) some form of invariants
- WP naturally takes into account assumptions `//@ assume`

solution VA exports its invariants on the form of WP-assumptions

Should work for any VA and WP engine

```
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;
    else
        res = 0;
    //l1: res == 0
}
```

```
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
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0
}
```

```
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
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0      // VA  $\oplus$  WP succeeds
}
```

Exported invariants

- numerical constraints (sets, intervals, congruence)
- only names appearing in the program (params, lhs, vars)
- in practice : **exhaustive export has very low overhead**

Soundness ok as long as VA is sound

Exhaustivity of “export” only affect deductive power

Summary

	VA	WP	VA \oplus WP
sound for assert validity	✓	✓	✓
blackbox reuse	✓	✓	✓
local precision	✗	✓	✓
calling context	✓	✗	✓
calls / loop effects	✓	✗	✓
global precision	✗	✗	✗
scalability wrt. #labels	✓	✓	✓
scalability wrt. code size	✗	✓	?

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**

	#Lab	#Inf	VA		WP		VA \oplus WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	60%	118	98%
Min		0	0	0%	0	0%	2	67%
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	82%	4.5	95%

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels

Detection power

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**

	#Lab	#Inf	VA		WP		VA \oplus WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	60%	118	98%
Min		0	0	0%	0	0%	2	67%
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	82%	4.5	95%

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels

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

report more accurate coverage ratio

Detection method	Coverage ratio reported by DSE*		
	None	VA \oplus 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

optimisation : speedup test generation : take care !

Ideal speedup

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

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

optimisation : speedup test generation : take care !

Ideal speedup

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

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

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

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 $a \wedge (b \vee c)$:

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

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

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

. [no exact encoding is known for MCDC]

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

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

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%

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

Next :

- more criteria (MCDC, strong mutations)
- scale up infeasibility detection
- scale up DSE, better handling of functions [big challenge]