Sound and Quasi-Complete Detection of Infeasible Test Requirements

Sébastien Bardin

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

joint work with:
Mickaël Delahaye, Robin David, Nikolai Kosmatov,
Mike Papadakis, Yves Le Traon, Jean-Yves Marion

Context : white-box 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

- generate tests, decide when to stop, assess quality of testing
- definition : systematic way of deriving test requirements

Context : white-box 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

- generate tests, decide when to stop, assess quality of testing
- definition : systematic way of deriving test requirements

The enemy : Infeasible test requirements

- waste generation effort, imprecise coverage ratios
- cause : structural coverage criteria are ... structural
- detecting infeasible test requirements is undecidable

Recognized as a hard and important issue in testing

- no practical solution, not so much work [compared to test gen.]
- **real pain** [ex : aeronautics, mutation testing]

Our goals and results

Focus on white-box (structural) coverage criteria

Goals : automatic detection of infeasible test requirements

- *sound* method [thus, incomplete]
- applicable to a large class of coverage criteria
- strong detection power, reasonable detection speed
- rely as much as possible on existing verification methods

Our goals and results

Focus on white-box (structural) coverage criteria

Goals : automatic detection of infeasible test requirements

- *sound* method [thus, incomplete]
- applicable to a large class of coverage criteria
- strong detection power, reasonable detection speed
- rely as much as possible on existing verification methods

Results

- automatic, sound and generic method ✓
- new combination of existing verification technologies ✓
- experimental results : strong detection power [95%], reasonable detection speed [$\leq$ 1s/obj.], improve test generation ✓

Our goals and results

Focus on white-box (structural) coverage criteria

Goals : automatic detection of infeasible test requirements

- *sound* method [thus, incomplete]
- applicable to a large class of coverage criteria
- strong detection power, reasonable detection speed
- rely as much as possible on existing verification methods

Results

- automatic, sound and generic method ✓
- new combination of existing verification technologies ✓
- experimental results : strong detection power [95%], reasonable detection speed [$\leq$ 1s/obj.], improve test generation ✓
- yet to be proved : scalability on large programs ?

  [promising, not yet end of the story]

Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Focus : Labels [ICST 2014]

- 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

Focus : Labels [ICST 2014]

- 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
  - infeasible label $(loc, \varphi) \Leftrightarrow$ valid assertion $(loc, \texttt{assert} \neg\varphi)$

Infeasible labels, valid assertions

```
int g(int x, int a) {
  int res;
  if(x+a >= x)
    res = 1;
  else
    res = 0;
 //l1: res == 0     // infeasible
}
```

Infeasible labels, valid assertions

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

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

- ✓ **IC**, **DC**, **FC**
- ✓ **CC**, **DCC**, **MCC**, **GACC**
- ✓ large part of Weak Mutations

✓ : perfect simulation [ICST 14]

Simulation of standard coverage criteria

- ✓ **IC**, **DC**, **FC**
- ✓ **CC**, **DCC**, **MCC**, **GACC**
- ✓ large part of Weak Mutations
- ≈ Strong Mutations
- ≈ MCDC

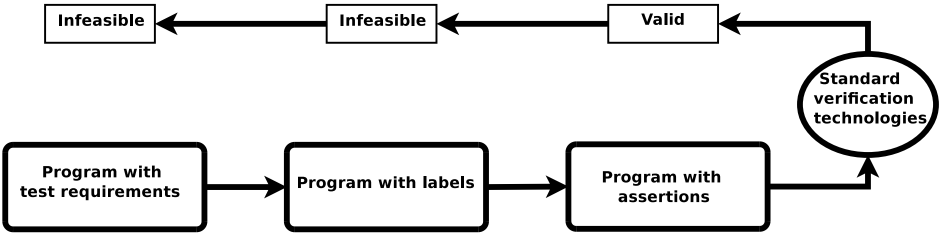
✓ : perfect simulation [ICST 14]
≈ : approx. simulation

Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Overview of the approach

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



- only soundness is required (verif)
  - label encoding not required to be perfect

Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Focus : checking assertion validity

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 [$pre^{\leq k}$, weakest precond., cegar]
  - perform a dedicated check for each assertion
  - a single check usually easier, but many of them

Focus : checking assertion validity

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 [pre$^{\leq k}$, weakest precond., 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
}
```

Proposal : VA $\oplus$ WP (1)

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**

Proposal : VA $\oplus$ WP (1)

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

VA⊕WP succeeds !

```
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⊕WP succeeds !

```
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⊕WP succeeds !

```
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 ⊕ WP succeeds
}
```

Proposal : VA $\oplus$ WP (2)

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	✗	✓	**?**

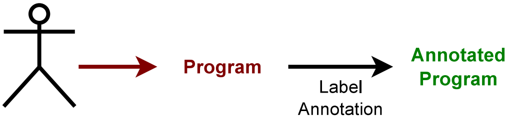
Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Implementation

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

Implementation inside LTest [TAP 14]

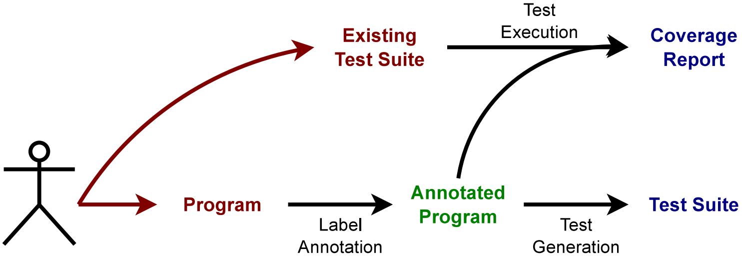


Supported criteria

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

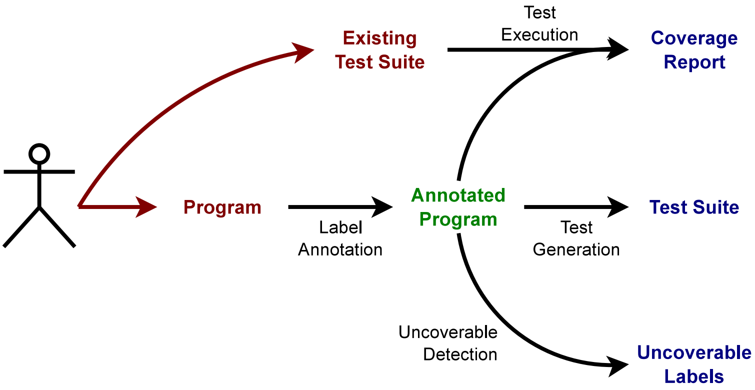
Encoded with labels [ICST 2014]

- managed in a unified way
- rather easy to add new ones



DSE* procedure [ICST 2014]

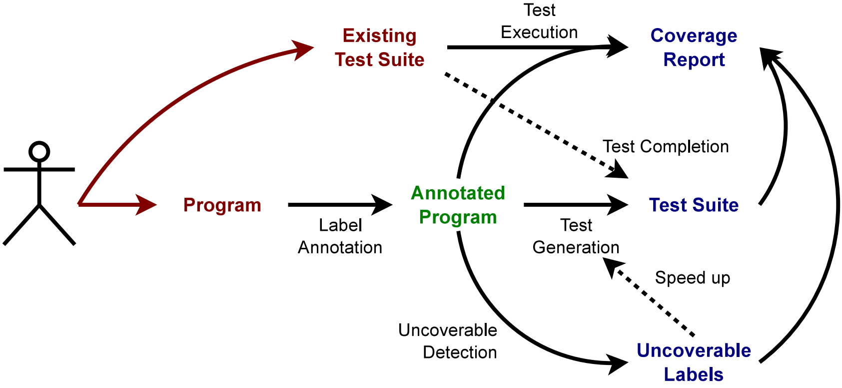
- DSE with native support for labels
- extension of PATHCRAWLER



Reuse static analyzers from FRAMA-C

- sound detection !
- several modes : VA, WP, VA $\oplus$ WP

Implementation inside LTEST [TAP 14]



Reuse static analyzers from FRAMA-C

- sound detection !
- several modes : VA, WP, VA ⊕ WP

Service cooperation

- share label statuses
- `Covered`, `Infeasible`, `?`

Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Experiments

RQ1 : How effective are the static analyzers in detecting infeasible test requirements ?

RQ2 : How efficient are the static analyzers in detecting infeasible test requirements ?

RQ3 : To what extent can we improve test generation by detecting infeasible test requirements ?

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, 121 infeasible ones

RQ1 : detection power

	#Lab	#Inf	VA		WP		VA ⊕ 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

RQ1 : detection power

	#Lab	#Inf	VA		WP		VA ⊕ 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

- clearly, VA ⊕ WP better than VA or WP alone
- VA ⊕ WP achieves almost perfect detection
- results from WP should scale

RQ2 : detection speed

Three usage scenarios

- a priori : all labels [before testing]
- a posteriori : those not covered by DSE* [after thorough testing]
- mixed : those not covered by RT [after cheap testing]

scenario	**#Lab**	**VA**	**WP**	**VA ⊕WP**
a priori	1,270	21.5	994	1,272
mixed	480	20.8	416	548
a posteriori	121	13.4	90.5	29.4

RQ2 : detection speed

Three usage scenarios

- a priori : all labels [before testing]
- a posteriori : those not covered by DSE* [after thorough testing]
- mixed : those not covered by RT [after cheap testing]

scenario	#Lab	VA	WP	VA $\oplus$WP
a priori	1,270	21.5	994	1,272
mixed	480	20.8	416	548
a posteriori	121	13.4	90.5	29.4

- VA mostly indep. from #Lab, WP linear,
  VA $\oplus$ WP in between
- good news : $\leq$ 1s per label, cost decreased by cheap testing

RQ3 : Impact on test generation

Impact 1 : report more accurate coverage ratio

	Coverage ratio reported by DSE⋆				
Detection method	**None**	**VA**	**WP**	**VA ⊕WP**	**Perfect***
Total	**90.5%**	96.9%	95.9%	**99.2%**	100.0%
Min	**61.54%**	80.0%	67.1%	**91.7%**	100.0%
Max	100.00%	100.0%	100.0%	100.0%	100.0%
Mean	**91.10%**	96.6%	97.1%	**99.2%**	100.0%

* preliminary, manual detection of infeasible labels

RQ3 : Impact on test generation

Impact 2 : speedup test generation

		VA	WP	VA ⊕ WP
		Speedup	**Speedup**	**Speedup**
RT(1s) +LUNCOV +DSE*	**Total**	**2.4x**	**2.2x**	**2.2x**
	Min	0.5x	0.1x	0.1x
	Max	**107.0x**	**74.1x**	**55.4x**
	Mean	**7.5x**	**5.1x**	**3.8x**

RT : random testing
Speedup wrt. DSE* alone

RQ3 : Impact on test generation

- improvement 1 : better coverage ratio
  - avg. 91% min. 61% $\rightarrow$ avg. 99% min. 92%
- improvement 2 : speed up test generation, in some cases [beware !]
  - avg. 3.8$\times$, min. 0.1$\times$, max. 55.4$\times$

Outline

- Introduction
- Background : labels
- Overview of the approach
- Focus : checking assertion validity
- Implementation
- Experiments
- Conclusion

Discussion

Related work

- some work detect (branch) infeasibility as a by product
  [Beyer et al. 07, Beckman et al. 10, Baluda et al. 11]
- detection of (weakly) equivalent mutants [reach, infect] through compiler optimizations or CSP [Offutt et al. 94, 97]
- detection of (strongly) equivalent mutants [Papadakis et al. 2015]
  - good on propagation (40%), not so good on reach/infect
  - very complementary

Scalability [other threats : see article]

- as scalable as the underlying technologies
- especially, WP is scalable wrt. code size (currently, VA is not)

Conclusion

Challenge : detection of infeasible test requirements

Results

- automatic, sound and generic method ✓
  - rely on labels and a new combination VA $\oplus$ WP
- promising experimental results ✓
  - strong detection power [95%]
  - reasonable detection speed [$\leq$ 1s/obj.]
  - improve test generation [better coverage ratios, speedup]

Future work : scalability on larger programs

- confirm WP results on larger programs
- explore trade-offs of VA $\oplus$ WP