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 [≤ 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 [≤ 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
Annotate programs with **labels**

- predicate attached to a specific program instruction

Label \((loc, \varphi)\) is covered if a test execution

- reaches the instruction at \(loc\)
- satisfies the predicate \(\varphi\)

**Good for us**

- can easily encode a large class of coverage criteria [see after]
- in the scope of standard program analysis techniques
Annotate programs with **labels**
- predicate attached to a specific program instruction

Label \((loc, \varphi)\) is covered if a test execution
- reaches the instruction at \(loc\)
- satisfies the predicate \(\varphi\)

**Good for us**
- can easily encode a large class of coverage criteria [see after]
- in the scope of standard program analysis techniques
- infeasible label \((loc, \varphi) \iff \text{valid assertion } (loc, \text{assert } \neg \varphi)\)
Infeasible labels, valid assertions

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

```c
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

```c
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

Program with test requirements → Program with labels → Program with assertions

- 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
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** $[\text{pre}^{\leq k}, \text{weakest precond.}, \text{cegar}]$
  - 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)

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th>WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound for assert validity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blackbox reuse</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>local precision</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>calling context</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>calls / loop effects</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>global precision</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>scalability wrt. #labels</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>scalability wrt. code size</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

**hypothesis**: VA is interprocedural
VA and WP may fail

```c
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

```c
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
}
```
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
Proposal: VA ⊕ 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
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 ⊕ 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

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound for assert validity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blackbox reuse</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>local precision</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>calling context</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>calls / loop effects</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>global precision</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>scalability wrt. #labels</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>scalability wrt. code size</td>
<td>×</td>
<td>✓</td>
<td>?</td>
</tr>
</tbody>
</table>
Implementation inside \textsc{LTest} \cite{TAP14}

\begin{itemize}
\item plugin of the \textsc{Frama-C} analyser for C programs
  \begin{itemize}
  \item open-source
  \item sound, industrial strength
  \item among other: VA, WP, specification language
  \end{itemize}
\item \textsc{LTest} itself is open-source except test generation
  \begin{itemize}
  \item based on \textsc{PathCrawler} for test generation
  \end{itemize}
\end{itemize}
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
Implementation inside LTest [TAP 14]

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 ⊕ WP
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

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
</tr>
<tr>
<td>Total</td>
<td>1,270</td>
<td>84</td>
<td>69%</td>
<td>73</td>
<td>98%</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>15</td>
<td>29</td>
</tr>
<tr>
<td>Mean</td>
<td>4.7</td>
<td>3.2</td>
<td>63%</td>
<td>2.8</td>
<td>4.5</td>
</tr>
</tbody>
</table>

#d: number of detected infeasible labels
%d: ratio of detected infeasible labels
RQ1: detection power

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
</tr>
<tr>
<td>Total</td>
<td>1,270</td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>15</td>
</tr>
<tr>
<td>Mean</td>
<td>4.7</td>
<td>3.2</td>
<td>3.2</td>
<td>63%</td>
<td>2.8</td>
</tr>
</tbody>
</table>

#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]

<table>
<thead>
<tr>
<th>scenario</th>
<th>#Lab</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a priori</td>
<td>1,270</td>
<td>21.5</td>
<td>994</td>
<td>1,272</td>
</tr>
<tr>
<td>mixed</td>
<td>480</td>
<td>20.8</td>
<td>416</td>
<td>548</td>
</tr>
<tr>
<td>a posteriori</td>
<td>121</td>
<td>13.4</td>
<td>90.5</td>
<td>29.4</td>
</tr>
</tbody>
</table>
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]

<table>
<thead>
<tr>
<th>scenario</th>
<th>#Lab</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a priori</td>
<td>1,270</td>
<td>21.5</td>
<td>994</td>
<td>1,272</td>
</tr>
<tr>
<td>mixed</td>
<td>480</td>
<td>20.8</td>
<td>416</td>
<td>548</td>
</tr>
<tr>
<td>a posteriori</td>
<td>121</td>
<td>13.4</td>
<td>90.5</td>
<td>29.4</td>
</tr>
</tbody>
</table>

- VA mostly indep. from #Lab, WP linear,
  VA ⊕ WP in between
- good news : ≤ 1s per label, cost decreased by cheap testing
RQ3 : Impact on test generation

Impact 1 : report more accurate coverage ratio

<table>
<thead>
<tr>
<th>Detection method</th>
<th>None</th>
<th>VA</th>
<th>WP</th>
<th>VA + WP</th>
<th>Perfect*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>90.5%</td>
<td>96.9%</td>
<td>95.9%</td>
<td><strong>99.2%</strong></td>
<td>100.0%</td>
</tr>
<tr>
<td>Min</td>
<td><strong>61.54%</strong></td>
<td>80.0%</td>
<td>67.1%</td>
<td><strong>91.7%</strong></td>
<td>100.0%</td>
</tr>
<tr>
<td>Max</td>
<td>100.00%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Mean</td>
<td><strong>91.10%</strong></td>
<td>96.6%</td>
<td>97.1%</td>
<td><strong>99.2%</strong></td>
<td>100.0%</td>
</tr>
</tbody>
</table>

* preliminary, manual detection of infeasible labels
Impact 2: speedup test generation

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th>WP</th>
<th>VA $\oplus$ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speedup</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RT(1s)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LUnCOV</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DSE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2.4x</td>
<td>2.2x</td>
<td>2.2x</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>0.5x</td>
<td>0.1x</td>
<td>0.1x</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>107.0x</td>
<td>74.1x</td>
<td>55.4x</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>7.5x</td>
<td>5.1x</td>
<td>3.8x</td>
</tr>
</tbody>
</table>

RT: random testing
Speedup wrt. DSE* alone
RQ3: Impact on test generation

- **improvement 1**: better coverage ratio
  - avg. 91% min. 61% → avg. 99% min. 92%

- **improvement 2**: speed up test generation, in some cases
  [beware!]
  - avg. 3.8×, min. 0.1×, max. 55.4×
Outline

- Introduction
- Background: labels
- Overview of the approach
- Focus: checking assertion validity
- Implementation
- Experiments
- Conclusion
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 ⊕ WP
- promising experimental results ✓
  - strong detection power [95%]
  - reasonable detection speed [≤ 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 ⊕ WP