Incorrectness Proofs for Object-Oriented Programs via Subclass Reflection

Wenhua Li[1], Quang Loc Le[2], Yahui Song[1], Wei-Ngan Chin[1]

[1] National University of Singapore, Singapore

[2] University College London, United Kingdom

Over-approximating reasoning

- Hoare Logic (Hoare triple):

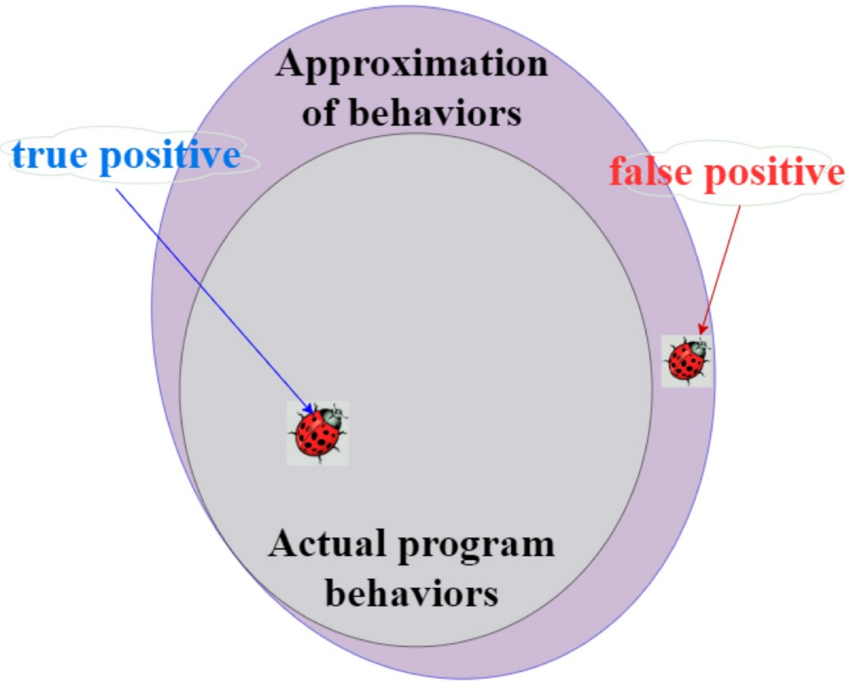
$$\{P\}\, c\, \{Q\} \qquad \textit{iff} \qquad post(c)P \subseteq Q$$

For all states s in P, if running c on s terminates in s', then s' is in Q.

- Prove the correctness of programs.
  - Q over-approximates post(c)P
  - Program behaviours are bounded by this triple.

Over-approximating reasoning

- Hoare Logic is imprecise for capturing bugs in programs



Under-approximating reasoning

- Incorrectness Logic (a dual theory of Hoare Logic)

Hoare triple:

$$\{P\}\, c\, \{Q\} \qquad \textit{iff} \qquad post(c)P \subseteq Q$$

Q over-approximates post(c)P

Under-approximate triple:

$$[P]\, c\, [Q] \qquad \textit{iff} \qquad post(c)P \supseteq Q$$

For all states s in Q, s can be reached by running $\boldsymbol{c}$ on some s' in P.

Q under-approximates post(c)P

Incorrectness Logic. Peter O'Hearn. POPL 2020

Under-approximating reasoning

- Incorrectness Logic (a dual theory of Hoare Logic)

Incorrectness triple

$$[P]\ c\ [\epsilon : Q]$$

ϵ: exit condition

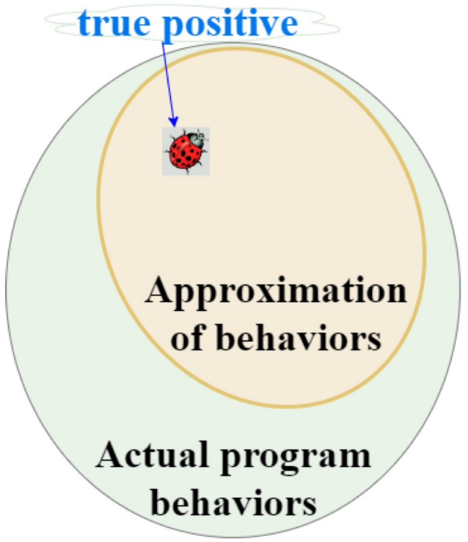
- [*ok*: normal execution]
- [*er*: erroneous execution]

- Dropping paths are allowed
- Every state in Q is a reachable state from some states in P

Incorrectness Logic. Peter O'Hearn. POPL 2020

Incorrectness logic

- A formal foundation for bug finding
- Incorrectness logic has been practically used for bug detection.
  - Pulse-X: an analyser based on Incorrectness Logic. It found 15 bugs which were unknown previously in OpenSSL.
  - Pulse: a commercial version.



Finding real bugs in big programs with incorrectness logic. Le et al., OOPSLA 2022

Incorrectness logic for OO programs

- Method calls in OOP:

```
f() {
    ...
    SomeClass a = new SomeClass(...);
    a.mth(...);
    ...
}
```

```
f(SomeClass a) {
    ...
    a.mth(...)
    ...
}
```

- The current approaches only support calls where where the called methods are determined statically.
- Features like Class inheritance, casting, and dynamic dispatching in OOP have not been studied yet.
- Many works have been done for correctness reasoning in OOP (e.g., Superclass abstraction, Class Invariant). There is no theoretical foundation for proving incorrectness in OOP.

Contributions

- Specification mechanism: a pair of specifications for each method.
  - Static specification: capturing the functional properties of a single class.
  - Reflexive specification: under-approximating the behaviours for one class and its superclasses (subclass reflection).

- Under-approximating proof system: verify the specifications.

- OURify (OO program Under-approximation Verifier) : an implementation which supports automated verification of specifications.

Illustrative example

- Cnt: tick() method increase *val* by 1
- DblCnt extends Cnt: tick() method stores the previous value of *val* in *bak* and non-deterministically increases *val* by 1 or 2

```
class Cnt {
  int val;
  void tick()
  {this.val := this.val+1;}}

class DblCnt extends Cnt{
  int bak;
  override void tick()
  {this.bak := this.val;
  if (*) super.tick();
  else
   this.val := this.val+2;}}
```

Illustrative example

```
class Cnt {
  int val;
  void tick()
  {this.val := this.val+1;}}

class DblCnt extends Cnt{
  int bak;
  override void tick()
  {this.bak := this.val;
  if (*) super.tick();
  else
   this.val := this.val+2;}}
```

- Static spec for Cnt.tick():

$[\mathit{this}::\texttt{Cnt}\langle v\rangle]\ \texttt{tick()}\ [\mathit{ok}\colon \mathit{this}::\texttt{Cnt}\langle v+1\rangle]$

- Static spec for DblCnt.tick():

$[\mathit{this}::\texttt{DblCnt}\langle v,b\rangle]\ \texttt{tick()}\ [\mathit{ok}\colon \mathit{this}::\texttt{DblCnt}\langle v',v\rangle \wedge v{+}1{\leq}v'{\leq}v{+}2]$

- Can be used for calls like:

```
Cnt a = new Cnt(...); ... a.tick();
DblCnt a = new BblCnt(...); ... a.tick();
```

- Can we efficiently reason about the following call?

```
f(Cnt a) {...a.tick();...}
```

Illustrative example

- OOP design should adhere to *Liskov substitution principle (behavioural subtyping)*: *An object of a subclass can always replace an object of the superclass without causing problems.*

- Based on LSP, we observe that a behavioural subtype should reflect the **reachable states** of its superclasses.

```
class Cnt {
  int val;
  void tick()
  {this.val := this.val+1;}}

class DblCnt extends Cnt{
  int bak;
  override void tick()
  {this.bak := this.val;
  if (*) super.tick();
  else
   this.val := this.val+2;}}
```

Illustrative example

- We propose dynamic view to encode multiple classes (disjuncts) simultaneously.

$$\begin{aligned} & \mathit{this}{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle \\ = \; & \mathit{this}{::}\texttt{Cnt}\langle v\rangle \;\vee\; \mathit{this}{::}\texttt{DblCnt}\langle v, b\rangle \end{aligned}$$

- The disjuncts can be merged iff the subclasses maintain the states for fields inherited from the superclasses

- Dynamic view: used in reflexive specs to support dynamic dispatching calls.

```
class Cnt {
  int val;
  void tick()
  {this.val := this.val+1;}}

class DblCnt extends Cnt{
  int bak;
  override void tick()
  {this.bak := this.val;
  if (*) super.tick();
  else
   this.val := this.val+2;}}
```

Illustrative example

- As Cnt has no superclass, its reflexive specs only reflects itself

  $\texttt{static/reflex}\ [\mathit{this}{::}\texttt{Cnt}\langle v\rangle]\ \texttt{tick}()\ [\mathit{ok}{:}\ \mathit{this}{::}\texttt{Cnt}\langle v+1\rangle]$

- DblCnt needs to reflect itself and Cnt

  $\texttt{static}\ [\mathit{this}{::}\texttt{DblCnt}\langle v, b\rangle]\ \texttt{tick}()\ [\mathit{ok}{:}\ \mathit{this}{::}\texttt{DblCnt}\langle v', v\rangle \wedge v{+}1{\leq}v'{\leq}v{+}2]$
  $\texttt{reflex}\ [\mathit{this}{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle]\ \texttt{tick}()\ [\mathit{ok}{:}\ \mathit{this}{::}\texttt{Cnt}\langle v{+}1\rangle\texttt{DblCnt}\langle v\rangle]$

  - The disjunct for else branch has been dropped here.

- We could also capture the else branch by using:

  $[\mathit{this}{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle]\_\ [\mathit{ok}{:}\ \mathit{this}{::}\texttt{DblCnt}\langle v{+}2, v\rangle]$

```
class Cnt {
  int val;
  void tick()
  {this.val := this.val+1;}}

class DblCnt extends Cnt{
  int bak;
  override void tick()
  {this.bak := this.val;
  if (*) super.tick();
  else
   this.val := this.val+2;}}
```

Illustrative example

Dynamic dispatching call

```
void goo(Cnt x) {...
```

$[x{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle]$

```
x.tick();
```

$[ok\colon\ x{::}\texttt{Cnt}\langle v+1\rangle\texttt{DblCnt}\langle v\rangle]$

```
y := (DblCnt) x;
```

$[ok\colon\ x{::}\texttt{DblCnt}\langle v+1, v\rangle \wedge y = x]$

$[er\colon\ x{::}\texttt{Cnt}\langle v+1\rangle]$

```
...}
```

A casting error

Verification of static specification

- Perform IL-style forward verification using the proof rules.

$$\frac{}{[x.f\mapsto e \land y = y']\ y{:=}x.f\ [ok\!: x.f\mapsto e[y'/y] \land y = e[y'/y]]}\ \text{Read}$$

$$\frac{}{[x = \mathtt{null}]\ y{:=}x.\_\ [er\!: x = \mathtt{null}]}\ \text{NullRead}$$

$$\frac{}{[x.f\mapsto e]\ x.f{:=}\ y\ [ok\!: x.f\mapsto y]}\ \text{Write} \qquad \frac{}{[x{=}\mathtt{null}]\ x.f{:=}y\ [er\!: x{=}\mathtt{null}]}\ \text{NullWrite}$$

$$\frac{}{[x = \mathtt{null} \land y = y']\ y{:=}x\ \mathtt{instanceof}\ C\ [ok\!: x = \mathtt{null} \land y = \mathit{False}]}\ \text{InsNull}$$

$$\frac{Q_1 \equiv x : C_1 \land y = \mathit{True} \land C_1 \prec C \qquad Q_2 \equiv x : C_1 \land y = \mathit{False} \land C_1 \not\prec C}{[x : C_1 \land y = y']\ \ y{:=}x\ \mathtt{instanceof}\ C\ [ok\!: Q_i],\ \ i \in \{1;2\}}\ \text{Ins}$$

$$\frac{\begin{array}{c} Q_1 \equiv x :: C_i\langle\overline{e_m}, \overline{e_i}\rangle C_k \land y = \mathit{True} \land C_i \prec C \\ Q_2 \equiv x :: C_m C_i\langle\overline{e_i}\rangle \land y = \mathit{False} \land C_i \not\prec C \end{array}}{[x :: C_m C_i\langle\overline{e_i}\rangle C_k \land y = y']\ \ y{:=}x\ \mathtt{instanceof}\ C\ [ok\!: Q_1 \lor Q_2]}\ \text{DyIns}$$

$$\frac{}{[x = \mathtt{null} \land y = y']\ y{:=}(C)\ x\ [ok\!: x = \mathtt{null} \land y = \mathtt{null}]}\ \text{CastNull}$$

$$\frac{}{[x{\mapsto}C_1\langle\bar{e}\rangle \land y = y' \land C_1{\prec}C]\ y{:=}(C)\ x\ [ok\!: x{\mapsto}C_1\langle\bar{e}[y'/y]\rangle \land y = x \land C_1{\prec}C]}\ \text{CastOk}$$

$$\frac{}{[x : C_1 \land C_1 \not\prec C]\ y{:=}(C)\ x\ [er\!: x : C_1 \land C_1 \not\prec C]}\ \text{CastErr}$$

$$\frac{Q \equiv x :: (C_i\langle\overline{e_m}, \overline{e_i}\rangle C_k)[y'/y] \land y = x \land C_i \prec C}{[x :: C_m C_i\langle\overline{e_i}\rangle C_k \land y = y']\ y{:=}(C)\ x\ [ok\!: Q]}\ \text{DyCastOk}$$

$$\frac{Q \equiv x :: C_m C_i\langle\overline{e_i}\rangle \land y = y' \land C_i \not\prec C}{[x :: C_m C_i\langle\overline{e_i}\rangle C_k \land y = y']\ y{:=}(C)\ x\ [er\!: Q]}\ \text{DyCastErr}$$

$$[x = \mathit{null}]\ x.mn(\bar{y})\ [er\!: x = \mathit{null}]\ \text{Null MethodInv}$$

$$\frac{x : C \qquad \mathtt{static}(C.\mathtt{mn}(\bar{w})) = [Pr]\_[\epsilon{:}Po] \qquad Pr[x, \bar{z}/\mathit{this}, \bar{w}] \Rightarrow P}{[P \land y = y']y = x.\mathtt{mn}(\bar{z})[\epsilon{:}Po[x, \bar{z}, y/\mathit{this}, \bar{w}, \mathit{ret}]]}\ \text{Static MethodInv}$$

$$\frac{\begin{array}{c} \mathit{view}(x) = x :: ...D\langle...\rangle \\ \mathtt{reflex}(D.\mathtt{mn}(\bar{w})) = [Pr]\_[\epsilon{:}Po] \qquad Pr[x, \bar{z}/\mathit{this}, \bar{w}] \Rightarrow P \end{array}}{[P \land y = y']y = x.mn(\bar{z})[\epsilon{:}Po[x, \bar{z}, y/\mathit{this}, \bar{w}, \mathit{ret}]]}\ \text{Dynamic MethodInv}$$

$$\frac{\mathtt{static}(C(\bar{w})) = [Pr]\_[\epsilon{:}Po] \qquad Pr[\bar{y}/\bar{w}] \Rightarrow P}{[P \land x = x']x{:=}\mathit{new}\ C(\bar{y})[\epsilon{:}Po[\bar{y}, x/\bar{w}, \mathit{this}]]}\ \text{Constructor}$$

Verification of reflexive specification

- The reflexive specs are validated without verifying against method bodies.

- Perform specification subtyping checking

$$\frac{Q_D \wedge type(this) \in T_C \models Q_C * F \qquad F * P_C \models P_D}{[P_C]\ \_\ [\epsilon{:}Q_C] \ <:_U\ [P_D]\ \_\ [\epsilon{:}Q_D]}$$

- This relation is a corollary of IL consequence rule and the frame rule of Separation Logic.

- Given $[P_C]\ \_\ [\epsilon{:}Q_C]$ for superclass C and $[P_D]\ \_\ [\epsilon{:}Q_D]$ for subclass D and the relation holds, every program satisfying $[P_C]\ \_\ [\epsilon{:}Q_C]$ will satisfy $[P_D]\ \_\ [\epsilon{:}Q_D]$

Verification for OOP methods

$$\frac{\begin{array}{c}\mathtt{sp}{=}[P]_{\_}[\epsilon{:}Q] \\ [P]\ \mathsf{S};\mathtt{return\ y}\ [\epsilon{:}Q] \qquad (\mathit{Spec\ verification})\end{array}}{\mathit{virtual}\ \mathtt{t_1\ mn}\ (\bar{\mathtt{t}}_2\ \overline{x})\ [\mathit{static}\ \mathtt{sp}]\ [\mathit{reflexive}\ \mathtt{sp}]\ \{\mathsf{S};\mathtt{return}\ y\}\ \mathtt{in}\ C}$$

$$\frac{\begin{array}{c}D{\prec_1}C \quad \mathtt{sp_c}{=}\mathtt{static}(C.\mathtt{mn}) \quad \mathtt{rp_c}{=}\mathtt{reflex}(C.\mathtt{mn}) \quad \mathtt{sp'_c}{=}\mathtt{sp_c}[\mathtt{this}:D/\mathtt{this}:C] \\ \mathit{Compatible}(\mathtt{sp'_c},\mathtt{sp}) \qquad (\mathit{Spec\ verification}) \\ \mathtt{sp} <:_U \mathtt{rp} \qquad (\mathit{Dynamic\ Dispatch}) \\ \mathtt{rp_c} <:_U \mathtt{rp} \qquad (\mathit{Behavioural\ subtyping})\end{array}}{\mathit{inherit}\ \mathtt{t_1\ mn}\ (\bar{\mathtt{t}}_2\ \overline{x})\ [\mathit{static}\ \mathtt{sp}]\ [\mathit{reflexive}\ \mathtt{rp}]\ \{\}\ \mathit{in}\ D}$$

$$\frac{\begin{array}{c}D{\prec_1}C \quad \mathtt{rp_c}{=}\mathtt{reflex}(C.\mathtt{mn}) \quad \mathtt{sp}{=}[P]_{\_}[\epsilon{:}Q] \\ [P]\ \mathsf{S};\mathtt{return}\ y\ [\epsilon{:}Q] \qquad (\mathit{Spec\ verification}) \\ \mathtt{sp} <:_U \mathtt{rp} \qquad (\mathit{Dynamic\ Dispatch}) \\ \mathtt{rp_c} <:_U \mathtt{rp} \qquad (\mathit{Behavioural\ Subtyping})\end{array}}{\mathit{override}\ \mathtt{t_1\ mn}\ (\bar{\mathtt{t}}_2\ \overline{x})\ [\mathit{static}\ \mathtt{sp}]\ [\mathit{reflexive}\ \mathtt{rp}]\ \{\mathsf{S};\mathtt{return}\ y\}\ \mathtt{in}\ D}$$

Implementation and Evaluation

- OURify consists 10,000 lines of OCaml codes.
- Benchmarks are manually constructed or selected from public dataset. We only keep the crucial parts

Benchmark	LOC	TIME(s)	LoSpec	SUCCESS	FAILED
NPE_1	34	0.249	3	3	0
M_OK_2	61	0.815	8	6	2
M_NPE_3	60	0.811	9	9	0
M_CAST_4	79	0.695	13	11	2
M_OK_5	80	0.799	7	7	0
NPE_6	80	0.956	8	8	0
NPE_7	150	2.850	28	28	0
NPE_8	167	3.251	22	21	1
CAST_9	187	1.717	18	18	0
M_NPE&CAST_10	203	1.801	19	19	0
OK_11	321	5.418	49	43	6
NPE_12	331	4.907	42	38	4
NPE_13	335	5.962	53	53	0
M_NPE&CAST_14	524	9.498	84	84	0
NPE_15	709	13.282	99	99	0
Sum	3321	53.011	462	447	15

OURify and Pulse

- Pulse is unable to report some bugs that are manifests, but these bugs can be verified in OURify.
- Pulse does not detect casting errors but OURify supports casting operator reasoning.

Benchmark	OK_OR	Cast_OR	NPE_OR	Manifest	NPE_PS	Confirmed	FP_PS	FN_PS
NPE_1	1	0	2	1	1	1	0	0
NPE_6	5	0	3	1	0	0	0	1
NPE_7	23	0	5	2	2	2	0	0
NPE_8	17	0	4	3	0	0	0	3
CAST_9	10	8	0	3	0	0	0	3
OK_11	43	0	0	0	0	0	0	0
NPE_12	37	0	1	1	1	0	1	1
NPE_13	40	0	13	12	8	5	3	7
NPE_15	75	0	24	11	9	8	1	3
Sum	251	8	52	34	21	16	5	18

Limitations and future directions

- We support automated verification of specification, but specifications need to be provided. Hence, the bug detection is not fully automated.

- One can apply bi-abduction based on our proof system to infer specifications automatically.

- An analogy of class invariant in Incorrectness Logic is yet to be discovered/designed. We think class (in)variant might advance incorrectness reasoning.

Summary

- Specification mechanism: a pair of specifications for each method.
- Under-approximating proof system: verify the specifications.
- OURify: can prove specifications automatically, some of which are not detectable by the state-of-the-art tool.