Incorrectness Proofs for Object-Oriented Programs via Subclass Reflection

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

[1] National University of Singapore, Singapore
{liwenhua, yahuis, chinwn}@comp.nus.edu.sg
[2] University College London, United Kingdom
loc.le@ucl.ac.uk

Abstract. Inheritance and method overriding are crucial concepts in object-oriented programming (OOP) languages. These concepts support a hierarchy of classes that reuse common data and methods. Most existing works for OO verification focus on modular reasoning in which they could support dynamic dispatching and thus efficiently enforce the Liskov substitution principle on behavioural subtyping. They are based on *superclass abstraction* to reason about the correctness of OO programs. However, techniques to reason about the incorrectness of OOP are yet to be investigated.
In this paper, we present a mechanism that 1) specifies the normal and abnormal executions of OO programs by using ok specifications and er specifications respectively; 2) verifies these specifications by a novel under-approximation proof system based on incorrectness logic that can support dynamic modularity. We introduce *subclass reflection* with dynamic views and an adapted subtyping relation for under-approximation. Our proposal can deal with both OOP aspects (e.g., behavioural subtyping and casting) and under-approximation aspects (e.g., dropping paths). To demonstrate how the proposed proof system can soundly verify the specifications, we prove its soundness, prototype the proof system, and report on experimental results. The results show that our system can precisely reason about the incorrectness of programs with OOP aspects, such as proving the presence of casting errors and null-pointer-exceptions.

1 Introduction

Proving the correctness and incorrectness of programs are two sides of a coin. On one side is Hoare logic, the pioneering formal system for correctness reasoning. Its central feature is Hoare triple, denoted by $\{pre\}$ `S` $\{post\}$ where *pre* and *post* are assertion formulae in some logic, and `S` is a program written in some programming languages. This triple means if we execute `S` starting from any program state σ (σ are valuations of program variables) satisfying *pre* and if it terminates, we will obtain program states σ' satisfying *post*. We refer σ' as reachable states from *pre*. This interpretation implies:

– *post* may be an over-approximation of reachable states, i.e., some of its states may satisfy *post* but do not correspond to a terminating execution associated

with a starting state satisfying *pre*. As such, Hoare logic is primarily used for correctness proving. Given a program `S`, a precondition *pre*, and an assertion *bad* representing buggy states, to prove that `S` is safe, we can show $\{pre\}$ `S` $\{post\}$ is valid and the *post* does not contain any *bad* states.
– Hoare logic cannot be used to prove the incorrectness of programs (i.e., confirming that `S` has a *bad* property by establishing $post \wedge bad$ that is satisfiable is inaccurate). This is due to an over-approximating *post* state.

Recently, O'Hearn completed the other side of the puzzle with incorrectness logic (IL) [22]. Its centrepiece is IL triple, the under-approximation counterpart of Hoare triple. An IL triple, written as $[pre]$ `S` $[post]$, states that each state of σ', that satisfies *post*, is a reachable state from executing `S` from one or more inputs satisfying *pre*. Given an IL triple $[pre]$ `S` $[post]$ and a buggy assertion *bad*, `S` has a bug if $post \wedge bad$ is satisfiable. With this, we can always find a counter-example whose input value(s) satisfy *pre* from which `S` goes *bad*. Notably, Pulse-X, a recent IL analyser [12], found 15 new bugs in OpenSSL and showed the importance of incorrectness reasoning for the industrial codebase.

Though IL is a significant advance to under-approximating reasoning, it is currently limited to static modularity and does not support dynamic modularity for object-oriented programming (OOP). OOP is one of the vital components in many imperative programming languages (e.g., Java, Scala and C#). An OO program is a collection of classes, each of which contains a set of fields and methods. Classes could be subclasses of others to form a class hierarchy. Methods of the superclass can be inherited or overridden by the subclass. The design of OOP must adhere to the Liskov substitution principle on behavioural subtyping [18]: An object of a subclass can always substitute an object of the superclass, and dynamic dispatching of a method is determined based on its actual type at the runtime. Most existing OO verification works focus on the support of dynamic modularity to enforce the substitutivity efficiently. While these works support correctness reasoning with *superclass abstraction* in Hoare logic (e.g., [9,10,14,15]) or its extension, separation logic (e.g., [5,20,24,25]), none focuses on the incorrectness of programs. Therefore, incorrectness reasoning in OO programs is worth investigating.

We introduce IL for OOP, with the following challenge: How to support dynamic modularity to enforce behavioural subtyping in under-approximation? A key observation is that the superclasses are unaware of the behaviours of extension fields in the subclasses. However, the subclasses can reflect the reachable states for fields inherited from the superclasses. Hence, the specifications of the subclass methods can be used to show the behaviours of the subclass itself and the superclasses. We call this *subclass reflection*.

In some prior works [5,25] on correctness reasoning, they propose the co-existence of static and dynamic specifications. A static specification (spec) specifies the functional properties of each method, while a dynamic spec can be used to verify dynamic dispatching; and the specification subtyping relation between static/dynamic specs ensures behavioural subtyping. Similar to the prior works, we propose static specs and reflexive specs to specify OO programs in

under-approximation: A static spec under-approximates a single method while a reflexive spec under-approximates methods of one class and its superclasses. Moreover, we propose dynamic views which can efficiently support *subclass reflection* and reason about casting operations. Our primary contributions are as follows.

- We present an under-approximate approach to OO verification. Our proposal extends incorrectness logic, with *subclass reflection* using dynamic views, to specify both normal and incorrect behaviours of OO programs.
- We introduce a proof system that supports dynamic modularity (including dynamic dispatching for class inheritance, casting operator and `instanceof` operator) and under-approximating reasoning via dropping paths and classes.
- We prototype the proposal in a verifier, called OURify (OO program Under-approximation Verifier), and demonstrate its capability of proving the incorrectness of OO programs, which is beyond the state-of-the-art.

Organization. Sect. 2 illustrates our proposal with examples. Sect. 3 presents the target language and the assertion language. The proof system and our approach to behaviour subtyping are shown in Sect. 4. Sect. 5 discusses our implementation OURify. Finally, Sect. 6 shows related work and concludes.

2 Motivation and Overview

We first explain the dynamic modularity problem and how existing proposals address it in correctness reasoning using Hoare logic and separation logic (Sect. 2.1). After that, in Sect. 2.2, we discuss the motivation of a novel foundation for incorrectness reasoning via incorrectness logic by highlighting the fundamental differences between Hoare logic and incorrectness logic. Afterwards, we informally describe our proposal for incorrectness reasoning.

2.1 Correctness Reasoning with Superclass Abstraction

When the type of an object is dynamically determined, is there a modular way to verify this object without explicitly considering all the method implementations? Liskov substitution principle answers this question: the subclass implementation must satisfy the specification of the superclass for each inherited or overridden method. This process requires re-verification as all subclasses need to be checked and could be polynomial in the numbers of inherited classes.

To avoid re-verification and enforce behavioural subtyping efficiently, prior works [5,25] suggest that each method has a pair of specs: a static spec for the verification of its implementation and a dynamic spec involving behaviour subtyping. Furthermore, a method's static spec is a subtype (written as $<:_O$) of its dynamic spec. A method's dynamic spec in the subclass is a subtype of the dynamic spec in its superclass. This mechanism enhances behavioural subtyping, such that the dynamic spec of a superclass's method abstracts (possibly over-approximating) behaviours of all its subclass methods. This is the

so-called *superclass abstraction*. Suppose that superclass C has a method `mn` with spec $\{pre_C\}\_\{post_C\}$, and D is a subclass of C – denoted as $D \prec C$, and D.`mn` overrides/inherits from C.`mn`. Then, for all D.`mn`'s spec $\{pre_D\}\_\{post_D\}$, $\{pre_D\}\_\{post_D\}$<:$_O$$\{pre_C\}_\{post_C\}$, where the relation <:$_O$ is defined as:

$$\frac{pre_C \wedge type(this) \prec D \models pre_D \qquad post_D \models post_C}{\{pre_D\}\_\{post_D\} <:_O \{pre_C\}\_\{post_C\}}$$

(Note that the relations proposed in separation logic [5,24,25] consider frame inference in the premises, which is a generic form of entailment problem.) Regarding this relation, we have the following two observations.

- First, the entailment checks are not straightforward, as the specs are from two different classes. Various approaches [8,25,5] have been applied to address this issue. For example, the extension predicate [5] encodes fields from multiple objects (e.g., one superclass and its subclasses) in a single predicate. When the extension predicate is used with the subtype constraint $type(this) \prec D$, the entailments are checked for the subclass D.
- Second, the subtyping relation enforces subtyping behaviour without requiring re-verification. For any program `S` s.t. $\{pre_D\}$`S`$\{post_D\}$ is valid, then the subtyping relation and the *consequence rule* of Hoare logic (rule `HL-Conseq` below) ensure so is $\{pre_C\}$`S`$\{post_C\}$.

$$\frac{pre_C \models pre_D \qquad \{pre_D\}\ \texttt{S}\ \{post_D\} \qquad post_D \models post_C}{\{pre_C\}\ \texttt{S}\ \{post_C\}}\ (\texttt{HL-Conseq})$$

We notice a phenomenon in which inheritance is not subtyping [7], i.e. subclass instances behave differently from instances of its superclass. One solution to address such odd instances is to provide over-approximation for *superclass abstraction* (shown in the following example). Alternatively, Dhara and Leavens [8] propose a *specification inheritance* technique in which the specification of the overriding method is strengthened by conjoining it with the specification of the overridden method. This technique was realized in separation logic via multiple specs [5] or specs with the *also* keyword [25].

We elaborate on subtyping behaviour through the code shown in Fig. 1. It defines two classes: the superclass `Cnt` and the subclass `DblCnt`. `Cnt` includes a field `val` and a method `tick`, which increases `val` by one. `DblCnt` inherits `val`, defines another field `bak` and overrides the method `tick`. Method `DblCnt.tick()` additionally backs up the value of `val` in `bak` and nondeterministically increases `val` by 1 (on line 10) or 2 (on line 12).

```
1  class Cnt {
2    int val;
3    void tick()
4    {this.val := this.val+1;}}
5
6  class DblCnt extends Cnt{
7    int bak;
8    override void tick()
9    {this.bak := this.val;
10   if (*) super.tick();
11   else
12    this.val := this.val+2;}}
```

Fig. 1. Illustrative example

While the `if` branch shows the subtyping behaviour of `DblCnt.tick()`, the `else` branch does not.

To write method specifications, we need to define an abstraction that captures all fields of the two classes. For instance, we follow the approach introduced in [5] to define an extension predicate in separation logic. The abstraction is:

$$this{::}\texttt{Cnt}\langle t, v, p\rangle * p{::}\texttt{ExtAll}(\texttt{Cnt}, t)$$

$this{::}\texttt{Cnt}\langle t, v, p\rangle$ defines the superclass. t is the actual type of $this$ object; value v is the field `val`. p is the reference to subclass extensions. $*$ is the separating conjunction, and predicate $p{::}\texttt{ExtAll}(\texttt{Cnt}, t)$ defines a chain of subclasses from `Cnt` to t. $\texttt{ExtAll}(\texttt{Cnt}, t)$ is defined as the following:

$$\begin{aligned} p :: \texttt{ExtAll}(\texttt{Cnt}, t) \equiv\ & t = \texttt{Cnt} \wedge p = null \\ & \vee\ p{::}\texttt{Ext}\langle t_1, \overline{v}, p_1\rangle * p_1{::}\texttt{ExtAll}(t_1, t) \wedge (t_1 \prec_1 \texttt{Cnt}) \wedge (t \prec t_1) \end{aligned}$$

Where $t_1 \prec_1 \texttt{Cnt}$ means t_1 is an immediate subclass of `Cnt` and $t \prec t_1$ means t is a subclass of t_1. With this abstraction, `Cnt` is realized as:

$$this{::}\texttt{Cnt}\langle \texttt{Cnt}, v, p\rangle * p{::}\texttt{ExtAll}(\texttt{Cnt}, \texttt{Cnt}) = this{::}\texttt{Cnt}\langle \texttt{Cnt}, v, null\rangle$$

And `DblCnt` is $this{::}\texttt{Cnt}\langle \texttt{DblCnt}, v, p\rangle * p{::}\texttt{ExtAll}(\texttt{Cnt}, \texttt{DblCnt})$ which is equivalent with $this{::}\texttt{Cnt}\langle \texttt{DblCnt}, v, p\rangle * p{::}\texttt{Ext}\langle \texttt{DblCnt}, b, null\rangle \wedge \texttt{DblCnt} \prec_1 \texttt{Cnt}$.

Using these predicates, we can write static and dynamic specs for two methods `tick`. First, methods `Cnt.tick` and `DblCnt.tick` are specified and statically verified by the following two static specs, respectively.

$$\texttt{static}\ \ \{this{::}\texttt{Cnt}\langle \texttt{Cnt}, v, null\rangle\}\ \texttt{Cnt.tick}()\ \{this{::}\texttt{Cnt}\langle \texttt{Cnt}, v+1, null\rangle\}$$

$$\begin{aligned} \texttt{static}\ \{this{::}\texttt{Cnt}\langle \texttt{DblCnt}, v, p\rangle * p{::}\texttt{Ext}\langle \texttt{DblCnt}, \_, null\rangle \wedge \texttt{DblCnt} \prec_1 \texttt{Cnt}\} \\ \texttt{DblCnt.tick}() \\ \{this{::}\texttt{Cnt}\langle \texttt{DblCnt}, v', p\rangle * p{::}\texttt{Ext}\langle \texttt{DblCnt}, v, null\rangle \wedge \texttt{DblCnt} \prec_1 \texttt{Cnt} \\ \wedge\ v+1 \leq v' \leq v+2\} \end{aligned}$$

Similarly, each method `tick` is annotated with another dynamic spec, which is used for dynamic dispatching verification.

$$\begin{aligned} \texttt{dynamic}\quad & \{this{::}\texttt{Cnt}\langle t, v, p\rangle * p{::}\texttt{ExtAll}(\texttt{Cnt}, t)\} \\ & \texttt{Cnt.tick}() \\ & \{this{::}\texttt{Cnt}\langle t, v', p\rangle * p{::}\texttt{ExtAll}(\texttt{Cnt}, t) \wedge v' > v\} \end{aligned}$$

$$\begin{aligned} \texttt{dynamic}\quad & \{this{::}\texttt{Cnt}\langle t, v, p\rangle * p{::}\texttt{Ext}\langle \texttt{DblCnt}, \_, p_1\rangle \\ & *p_1{::}\texttt{ExtAll}(\texttt{DblCnt}, t) \wedge \texttt{DblCnt} \prec_1 \texttt{Cnt}\} \\ & \texttt{DblCnt.tick}() \\ & \{this{::}\texttt{Cnt}\langle t, v', p\rangle * p{::}\texttt{Ext}\langle \texttt{DblCnt}, v, p_1\rangle \\ & *p_1{::}\texttt{ExtAll}(\texttt{DblCnt}, t) \wedge \texttt{DblCnt} \prec_1 \texttt{Cnt} \wedge\ v{+}1 \leq v' \leq v{+}2\} \end{aligned}$$

Next, to enforce behaviour subtyping, we first check whether the static spec of method `Cnt.tick()` is a subtype of its dynamic spec. Secondly, we check

whether the dynamic spec of `DblCnt.tick()` is a subtype of the dynamic spec of `Cnt.tick()`. With these specs above, all these checks are valid. Hence, this validity guarantees behavioural subtyping without requiring re-verification. Moreover, any dynamic dispatching call with the receiver of static type `Cnt` can use the dynamic specification in class `Cnt`.

2.2 Incorrectness Reasoning with Subclass Reflection

Hoare logic and IL have different foundations. Technically, IL has another consequence rule with a reversed entailment in the premises.

$$\frac{pre_D \models pre_C \qquad [pre_D]\ \mathtt{S}\ [post_D] \qquad post_C \models post_D}{[pre_C]\ \mathtt{S}\ [post_C]}\ (\mathtt{IL-Conseq})$$

Second, an analyser using Hoare logic has to prove the safety of all program paths to show the absence of bugs in a program. In contrast, to show the presence of a bug, an analyser using IL could drop paths. A critical insight from the `IL-Conseq` rule is that the postcondition can be under-approximated, e.g., dropping paths/disjuncts for scalability. *Superclass abstraction* cannot be easily adapted to capture reachable states for subclasses in under-approximation. As the above example shows, the dynamic spec of `Cnt.tick` only records the change in the `val` field; we cannot conclude any information for the `bak` field. As a result, we cannot find precise reachable states for the subclass of `Cnt` when a dynamic dispatching call is performed.

We observe that while superclasses are unaware of reachable states of extended fields in the subclasses, the subclasses should satisfy the constraints (reachable states) over fields inherited from superclasses. To uphold the substitution principle in under-approximating reasoning, we require the inherited fields in the postcondition of a subclass method are not weaker than its counterpart in the superclass.

Based on this observation, we introduce *subclass reflection* to handle dynamic dispatching calls for under-approximating reasoning. *Superclass abstraction* is a top-down approach while *subclass reflection* is bottom-up. An abstraction for under-approximation could be behaviours of a subset of a class hierarchy. With this setting, we write reflexive specifications in subclasses to reflect their superclasses' behaviours. Hence, each subclass will take care of one class chain in a class hierarchy.

Given a subclass method $D.\mathtt{mn}$, for all $D.\mathtt{mn}$'s specs $[pre_D]\_[post_D]$, there exists some specs $[pre_C]\_[post_C]$ of method `mn` in the superclass C such that $[pre_C]\_[post_C] <:_U [pre_D]\_[post_D]$ where the relation $<:_U$ [3] is defined as:

$$\frac{pre_C \models pre_D \qquad post_D \wedge type(this) = C \models post_C}{[pre_C]\_[post_C] <:_U [pre_D]\_[post_D]}$$

If $[pre_C]\ \_\ [post_C] <:_U [pre_D]\ \_\ [post_D]$, then for all `S`, and $[pre_C]\ \mathtt{S}\ [post_C]$, we have $[pre_D]\ \mathtt{S}\ [post_D \wedge type(this) = C]$. Note that, the type constraint here

[3] This definition is slightly different from the version in Definition 2 for simplicity.

is $type(this) = C$ which is different from $type(this) \prec D$ in $<:_O$. This is because *subclass reflection* requires $post_D$ to reflect its superclass C. We now demonstrate our proposal through the example in Fig. 1. The reflexive spec of `Cnt.tick` is the same as its static spec since `Cnt` is the only type to be reflected by `Cnt`:

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

Note that, ok denotes postconditions in normal executions. Objects that need to be reflected by `DblCnt` are $this{::}\texttt{Cnt}\langle v\rangle \ \vee \ this{::}\texttt{DblCnt}\langle v, b\rangle$. We propose a dynamic view as: $this{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle$ to represent this disjunction.

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

The `else` branch of `DblCnt.tick` has been dropped in the reflexive spec.

Let `reflex(mn)` (resp. `static(mn)`) be the reflexive (resp. static) spec of method `mn`. To show that `reflex(DblCnt.tick)` is valid for both `DblCnt.tick` and `Cnt.tick`, we prove both 1) `static(DblCnt.tick)` $<:_U$ `reflex(DblCnt.tick)` and 2) `reflex(Cnt.tick)` $<:_U$ `reflex(DblCnt.tick)`. We illustrate 1) here,

$$\begin{array}{ll} & this{::}\texttt{DblCnt}\langle v, b\rangle \models this{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle \qquad //\text{checking for } pre\\ & this{::}\texttt{Cnt}\langle v{+}1\rangle\texttt{DblCnt}\langle v\rangle \wedge (type(this) = \texttt{DblCnt}) \qquad //\text{checking for } post\\ \Rightarrow & (this{::}\texttt{Cnt}\langle v{+}1\rangle \vee this{::}\texttt{DblCnt}\langle v{+}1, v\rangle) \wedge (type(this) = \texttt{DblCnt})\\ \Rightarrow & this{::}\texttt{DblCnt}\langle v{+}1, v\rangle \models this{::}\texttt{DblCnt}\langle v', v\rangle \wedge v{+}1{\leq}v'{\leq}v{+}2\end{array}$$

By doing so, we validate `reflex(DblCnt.tick)` without verifying it against the method bodies. We show a simple example in Fig. 2. The precondition before the dispatching call (line 3) shows that object x has a dynamic view $x{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle$. We will retrieve a reflexive spec according to the last type in this dynamic view. Hence, the reflexive spec in `DblCnt` is chosen as it also reflects the types before `DblCnt` in this dynamic view. Alternatively, if we want to capture the `else` branch of `DblCnt`, another reflexive spec in `DblCnt.tick()` could be: $[this{::}\texttt{Cnt}\langle v\rangle\texttt{DblCnt}\langle b\rangle]\_\ [ok{:}\ this{::}\texttt{DblCnt}\langle v{+}2, v\rangle]$. This reflexive spec drops the path from `Cnt`. Hence, we do not have to check `reflex(Cnt.tick)` $<:_U$ `reflex(DblCnt.tick)` for this spec as the relation is trivially true.

Our dynamic view can reason about casting, which is extensively used in OOP. For instance, Fig. 2 shows a casting operation performed on object x. x's type is either `Cnt` or `DblCnt`. On line 5, as x is casting to `DblCnt`, based on x's possible types, our system splits into cases with ok spec on line 6 and er spec on line 7, respectively. By so doing, our system can discover bugs relating to casting effectively. The efficiency is also confirmed by our experiments: Our system can prove casting bugs which are beyond Pulse, the bug checker used in products at Meta and other big-tech companies.

```
1  void goo(Cnt x) {...
2       [x::Cnt⟨v⟩DblCnt⟨b⟩]
3       x.tick();
4       [ok: x::Cnt⟨v + 1⟩DblCnt⟨v⟩]
5       y := (DblCnt)  x;
6       [ok: x::DblCnt⟨v + 1, v⟩ ∧ y = x]
7       [er: x::Cnt⟨v + 1⟩]
8     ...}
```

Fig. 2. Example on casting

3 Language and Specifications

The section presents the core OO language and our assertion grammar.

3.1 Syntax of the Target Language

Fig. 3 presents our core language. We assume the language uses single inheritance and pass-by-value mechanism. `Object` is an implicit superclass of all classes, $x, y...$ for program variables, $C, D...$ for class names, e and B for expressions and boolean expressions respectively, and $x.f$ for the field f of x. Boolean expression x `instanceof` C is true if x is in class C or a subclass of C.

$$
\begin{array}{rl}
\mathscr{P} ::= & \overline{\texttt{cdef}}; \\
\texttt{cdef} ::= & \mathit{class}\ C_1\ \mathit{extends}\ C_2\ \{\overline{\texttt{t}}\ \overline{f};\ \overline{\texttt{meth}}\} \\
\tau ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{void} \qquad\qquad \texttt{t} ::= C \mid \tau \\
\texttt{sp}, \texttt{rp} ::= & [P]\_[\epsilon{:}Q] \\
\texttt{meth} ::= & \texttt{mtype t mn}\ (\overline{\texttt{t}}\ \overline{x})\ [\mathit{static}\ \texttt{sp}]\ [\mathit{reflexive}\ \texttt{rp}]\ \{\texttt{S};\texttt{return}\ y\} \\
\texttt{mtype} ::= & \mathit{virtual} \mid \mathit{inherit} \mid \mathit{override} \\
\texttt{S} ::= & \texttt{skip} \mid x{:=}e \mid x.f{:=}y \mid x{:=}y.f \mid \texttt{t}\ x;\texttt{S} \mid y{:=}(C)\ x \mid x{:=}\mathit{new}\ C(\overline{y}) \\
& \mid y{:=}x.\texttt{mn}(\overline{z}) \mid y{:=}x\ \texttt{instanceof}\ C \mid \texttt{S};\texttt{S} \mid \mathit{assume}(B) \mid \texttt{S} + \texttt{S}
\end{array}
$$

Fig. 3. A core Object-Oriented language.

A program $\mathscr{P}$ is a collection of class definitions. A class declares its superclass via keyword *extends*. A class consists of fields, method declarations and definitions. Each method `meth` will be annotated as *virtual*, *inherit* or *override*. A *virtual* method only exists in the subclass but not its superclass. An *inherit* method uses the same method body as its superclass. Lastly, an *override* method re-defines the method body in the subclass. Each method is annotated with two specifications: one is static `sp` and another is reflexive `rp`. ϵ is program status: ok (for normal executions) and er (for abnormal ones).

3.2 Semantics

Val defines values of variables including integers, booleans, locations *Loc*, and *null*. A program state $\sigma \in \mathit{PState}$ is a tuple, including a stack $s \in \mathit{Stack}$, that maps variables to values, *Val*, and a heap $h \in \mathit{Heap}$, that partially maps addresses to the contents. A heap h includes two mappings: $h.1$ maps locations to class names (dynamic type of an object) and $h.2$ maps location-field tuples to *Val*. The semantics is the relation of statements `S`, exit conditions ϵ, and program states σ.

$$
\begin{array}{lll}
\sigma \in \mathit{PState} \stackrel{\text{def}}{=} \mathit{Stack} \times \mathit{Heap} & s \in \mathit{Stack} \stackrel{\text{def}}{=} \mathit{Var} \rightarrow \mathit{Val} & v \in \mathit{Val} \\
h \in \mathit{Heap} \stackrel{\text{def}}{=} (\mathit{Loc} \rightharpoonup \mathit{Classes}) \times (\mathit{Loc} \times \mathit{Field} \rightharpoonup \mathit{Val}) & & l \in \mathit{Loc} \subseteq \mathit{Val} \\
[\![.]\!] \stackrel{\text{def}}{=} \mathit{Statement} \times \mathit{Exit} \times \mathscr{P}(\mathit{PState} \times \mathit{PState}) & \epsilon \in \mathit{Exit} \stackrel{\text{def}}{=} \{ok, er\} &
\end{array}
$$

The relational denotational semantics is presented in Appendix A – Fig. 6. We discuss the semantics of two commands: casting and `instanceof` in detail. The semantics of casting is as follows:

$$[\![y{:=}(C)\ x]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[y \mapsto s(x)],h))|\ (h.1(s(x)) = C_1 \wedge C_1 \prec C) \vee s(x) = null\}$$
$$[\![y{:=}(C)\ x]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ h.1(s(x)) = C_1 \wedge C_1 \not\prec C\}$$

Casting an object to its superclass is always successful, while it is erroneous another way around. For instance, downcasting a heap object with a type C to its subclass (not itself) or any unrelated class causes an error. The statement `instanceof` is used to check object types before casting.

Class hierarchy is collected via *extends* keyword. When for each C_1 *extends* C_2, $\{C_1 \prec C_2\}$ is added to the environment. We can query the class hierarchy environment for the subtyping relation between classes. This relation is reflexive and transitive. We use notations $C_2 \prec_1 C_1$ to mean C_2 is the immediate subclass of C_1. x `instanceof` C is a side-effect-free Boolean expression. Its semantics is as follows:

$$B[\![x\ \texttt{instanceof}\ C]\!](s,h) \stackrel{\text{def}}{=} False\ \texttt{iff}\ s(x) = null \vee (h.1(s(x)) \not\prec C)$$
$$B[\![x\ \texttt{instanceof}\ C]\!](s,h) \stackrel{\text{def}}{=} True\ \texttt{iff}\ (h.1(s(x)) \prec C)$$

3.3 Assertion Language

We here present the assertion language, an extension of separation logic [5] with IL. Fig. 4 presents the syntax of the specification language (while the semantics can be found in Appendix A - Fig. 7). The separation conjunction $\kappa_1 * \kappa_2$ describes two non-overlapping heaps, κ_1 and κ_2. $x.f{\mapsto}e$ stands for an object x has a field f which points to e and $x : C$ stands for the type for x stored in a heap is C. To simplify the notation, we encode a heap object in the form of $x{\mapsto}C\langle\overline{e}\rangle$, meaning that the object x of exact type C has fields $x.f_1{\mapsto}e_1, x.f_2{\mapsto}e_2, \cdots x.f_n{\mapsto}e_n$. That said, $x{\mapsto}C\langle\overline{e}\rangle = x : C * x.f_1{\mapsto}e_1 * x.f_2{\mapsto}e_2 * \cdots x.f_n{\mapsto}e_n$.

$$P, Q ::= (\kappa \wedge \phi) \mid P \vee Q \mid \exists x.P \qquad \kappa ::= emp \mid x.f{\mapsto}e \mid x : C \mid x{\mapsto}C\langle\overline{e}\rangle \mid \kappa_1 * \kappa_2$$
$$\phi ::= False \mid C \prec C \mid x = e \mid x < e \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \Rightarrow \phi_2$$

Fig. 4. Assertion language.

We also call $x{\mapsto}C\langle\overline{e}\rangle$ a static view, which describes a single object. In addition, we introduce the dynamic view to handle the dynamically dispatched method call. The dynamic view is in the form of $x{::}C_1\langle\overline{e_1}\rangle C_2\langle\overline{e_2}\rangle \cdots C_n\langle\overline{e_n}\rangle$ which is a collection of static views of objects along a class chain from C_1 to C_n in a class hierarchy. Specifically, it is syntactic sugar for the disjunction of objects, i.e. $x{\mapsto}C_1\langle\overline{e_1}\rangle \vee x{\mapsto}C_2\langle\overline{e_1}, \overline{e_2}\rangle \cdots \vee x{\mapsto}C_n\langle\overline{e_1}, \cdots \overline{e_n}\rangle$. The subclass objects have to maintain the same state for the fields inherited from its superclass to form a valid dynamic view.

IL triples. An IL triple is of the following form: $\vDash [P]\ \mathtt{S}\ [\epsilon{:}Q]$. In contrast to Hoare logic, the postcondition Q is an under-approximation of all possible execution paths and any state in Q, is reachable from some states satisfying P. Formally,

$$\vDash [P]\ \mathtt{S}\ [\epsilon{:}Q] \stackrel{\text{def}}{=} \forall \sigma \in [\![Q]\!].\ \exists \sigma' \in [\![P]\!].(\sigma', \sigma) \in [\![\mathtt{S}]\!]\epsilon.$$

4 Proof system for under-approximating reasoning

We propose specification subtyping in 4.1 and the mechanism of static and reflexive specifications in 4.2. Finally, proof rules are shown in 4.3.

4.1 Behavioural Subtyping

Liskov substitution principle (behaviour subtyping) [17,18] gives a general guideline for OOP design, which is crucial to the dynamic modularity problem. In under-approximation, we uphold this principle. Firstly, we define the specification subtyping for IL triples.

Definition 1 (Specification subtyping). *Given an IL specification* $[P_C]\_[\epsilon{:}Q_C]$ *and another IL specification* $[P_D]\_[\epsilon{:}Q_D]$. *We say* $[P_C]\_[\epsilon{:}Q_C]$ *is a subtype specification of* $[P_D]\_[\epsilon{:}Q_D]$ *if the following holds,*

$$\frac{Q_D \vDash Q_C * F \qquad F * P_C \ \vDash\ P_D}{[P_C]\_[\epsilon{:}Q_C] <: [P_D]\_[\epsilon{:}Q_D]}$$

This definition is a corollary of IL consequence rule and the frame rule. The frame F can be calculated via the postcondition entailment proving. Then, F will be carried forward for the precondition entailment proving. Any program satisfying $[P_C]\_[\epsilon{:}Q_C]$ will satisfy $[P_D]\_[\epsilon{:}Q_D]$.

Recap that the inherited fields in a behavioural subtype should not reach more states than the superclass. This is the key point to uphold Liskov substitution principle in under-approximation. For instance, we would not expect a buggy state to be reachable by a method in the subclass but unreachable in the superclass. Otherwise, the superclass is not replaceable as the program will introduce new errors with the subclass. Hence, according to the above definition, the under-approximation specification of a superclass should be a subtype of that in the subclass. In other words, the subclass specification needs to reflect its superclass's behaviours. We call it *subclass reflection*. With *subclass reflection*, the dynamic dispatching call can be handled efficiently.

However, as subclasses might extend superclasses with extra fields, checking Definition 1 is not straightforward. To address this issue, we incorporate static view and dynamic view. Recall that the dynamic view is a disjunction of multiple objects. We allow a constraint $type(x) \in T$ to assert if the type of object x is in a set T of types. Hence, we can check specifications for objects that only belong to T. For example,

$$\begin{aligned} & this{::}C\langle\overline{e_1}\rangle D\langle\overline{e_2}\rangle \wedge type(this) \in \{C\} \\ \Rightarrow & (this{\mapsto}C\langle\overline{e_1}\rangle \vee this{\mapsto}D\langle\overline{e_1}, \overline{e_2}\rangle) \wedge type(this) \in \{C\} \\ \Rightarrow & this{\mapsto}C\langle\overline{e_1}\rangle \end{aligned}$$

We mainly need two kinds of implications between static view and dynamic view in our verification processes 4.2.

Lemma 1 (View relationship).

$$this{::}C\langle\overline{e_1}\rangle D\langle\overline{e_2}\rangle E\langle\overline{e_3}\rangle \wedge type(this) \in \{E\} \Rightarrow this{\mapsto}E\langle\overline{e_1},\overline{e_2},\overline{e_3}\rangle$$

$$this{::}C\langle\overline{e_1}\rangle D\langle\overline{e_2}\rangle E\langle\overline{e_3}\rangle \wedge type(this) \in \{C,D\} \Rightarrow this{::}C\langle\overline{e_1}\rangle D\langle\overline{e_2}\rangle$$

Now, we introduce the specification subtyping for behavioural subtyping.

Definition 2 (Behavioural Subtyping). *We say that the under-approximation specification* $[P_C]\ \_\ [\epsilon{:}Q_C]$ *for a method* mn *in superclass* C *and another* $[P_D]\ \_\ [\epsilon{:}Q_D]$ *for* mn *in subclass* D *cater to behavioural subtyping if the following holds,*

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

where T_C *is the set of types pointed to by* `this` *reference in* C*'s specification. We use* $<:_U$ *to capture this relationship.*

4.2 Static and Reflexive Specifications

In some previous works [5,25], static and dynamic specification co-existence has been proposed to handle method verification and behavioural subtyping. We introduce a similar mechanism in an under-approximation flavour. We use the special variable *this* to denote the reference of the current object.

Static Specification Static specification is a description of a single method. The static view must describe the object referred to by *this* in the static specification. Hence, the static specification should be precise (the precondition needs to be as strong as possible, and the postcondition needs to be as weak as possible).

Reflexive Specification Reflexive specification is used for two purposes. Firstly, it ensures behavioural subtyping: i) The reflexive specification in the superclass is a subtype of the reflexive specification in the subclass; ii) the static specification of a method needs to be a subtype of the corresponding reflexive specification. Secondly, it is used for dynamically dispatching calls. To model dynamic dispatching, the dynamic views encode the state of multiple objects along a class chain. Any object in a dynamic view could be dispatched for a dynamic call. In contrast to the static specification, we use dynamic view for *this* reference in reflexive specifications.

Static/Reflexive Specification Verification. We now discuss the relationship between these two specifications in class inheritance. The first one is virtual method whose implementation only exists in subclasses. Note that, one specification can be both static and reflexive in the virtual method as there is no superclass to reflect.

$$\frac{\begin{array}{c}\texttt{sp}{=}[P]\_[\epsilon{:}Q] \\ [P]\ \texttt{S;return y}\ [\epsilon{:}Q] \qquad (\textit{Spec verification})\end{array}}{\textit{virtual}\ \texttt{t}_1\ \texttt{mn}\ (\overline{\texttt{t}}_2\ \overline{x})\ [\textit{static}\ \texttt{sp}]\ [\textit{reflexive}\ \texttt{sp}]\ \{\texttt{S;return}\ y\}\ \texttt{in}\ C}$$

Spec verification is the verification of the static specification against the method body by using our proof rules in Sec. 4.3 and Appendix C.

Second, an inherited method in the subclass uses the same implementation from its superclass: Inherited methods are semantically equivalent in both classes. The prior work [5] defines a notion called "statically-inherited" methods. A method is statically inherited by the subclass if (1) it does not override the original implementation and (2) if the method calls any other method `mn` inside the body, `mn` must also be statically-inherited. For simplicity, we assume every inherited method is statically-inherited as a non-statically-inherited method can always be transferred into an overriding method. To verify the static specification for the inherited method in the subclass, we can check whether its specification is *compatible* with the corresponding static specification in the superclass. $\mathit{Compatible}(sp', sp)$ means that sp is derivable from sp' using consequence rule or frame rule, i.e. $sp' <: sp$, which is defined in Definition 1.

$$\frac{\begin{array}{ccc}D{\prec_1}C & \texttt{sp}_\texttt{c}{=}\texttt{static}(C.\texttt{mn}) \quad \texttt{rp}_\texttt{c}{=}\texttt{reflex}(C.\texttt{mn}) & \texttt{sp}'_\texttt{c}{=}\texttt{sp}_\texttt{c}[\texttt{this}:D/\texttt{this}:C] \\ & \mathit{Compatible}(\texttt{sp}'_\texttt{c}, \texttt{sp}) \qquad (\textit{Spec verification}) & \\ & \texttt{sp} <:_U \texttt{rp} \qquad (\textit{Dynamic Dispatch}) & \\ & \texttt{rp}_\texttt{c} <:_U \texttt{rp} \qquad (\textit{Behavioural subtyping}) & \end{array}}{\textit{inherit}\ \texttt{t}_1\ \texttt{mn}\ (\overline{\texttt{t}}_2\ \overline{x})\ [\textit{static}\ \texttt{sp}]\ [\textit{reflexive}\ \texttt{rp}]\ \{\}\ \textit{in}\ D}$$

Note that, $\texttt{sp}'_\texttt{c} = \texttt{sp}_\texttt{c}[\texttt{this}:\texttt{D}/\texttt{this}:\texttt{C}]$ is valid when the superclass implementation does not access the type information of *this*. If the implementation accesses the type information of *this*, we need to verify the implementation against the static specification of the subclass.

Lastly, an overriding method redefines the procedure performed in a subclass. Hence, the superclass and the subclass may behave differently. However, the subclass can still be behavioural subtyping if both classes obey the rule in Definition 2. Again, we require the same relation holds for those specifications.

$$\frac{\begin{array}{ccc}D{\prec_1}C & \texttt{rp}_\texttt{c}{=}\texttt{reflex}(C.\texttt{mn}) & \texttt{sp}{=}[P]\_[\epsilon{:}Q] \\ [P]\ \texttt{S;return}\ y\ [\epsilon{:}Q] \qquad (\textit{Spec verification}) & & \\ \texttt{sp} <:_U \texttt{rp} \qquad (\textit{Dynamic Dispatch}) & & \\ \texttt{rp}_\texttt{c} <:_U \texttt{rp} \qquad (\textit{Behavioural Subtyping}) & & \end{array}}{\textit{override}\ \texttt{t}_1\ \texttt{mn}\ (\overline{\texttt{t}}_2\ \overline{x})\ [\textit{static}\ \texttt{sp}]\ [\textit{reflexive}\ \texttt{rp}]\ \{\texttt{S;return}\ y\}\ \texttt{in}\ D}$$

The constructor is a special type of method that initialises the fields of an object. We use C_{cs} to denote the constructor for class C. When the subclass's constructor is called, by default, the constructor of its superclass is called before the subclass constructor. As a constructor instantiates a concrete object, constructors only have a static specification. A concrete object should use static specifications for further method calls.

$$\frac{\begin{array}{c} D \prec_1 C \qquad \mathtt{sp_c}{=}\mathit{static}(C_{\mathtt{cs}}) \qquad \mathtt{sp_c}{=}[P_c]\_[\epsilon{:}Q_c] \\ \mathtt{sp}{=}[P]\_[\epsilon{:}Q] \qquad P_c * \mathtt{P_f} \vdash P \\ [P']{=}[\epsilon{:}Q_c[\mathtt{this} : D/\mathtt{this} : C] * \mathtt{P_f} * (*_{\mathtt{f_i}\in\mathtt{field(D)}}\, \mathtt{this.f_i}{\mapsto}\mathtt{null})] \\ [P']\ \mathtt{S}\ [\epsilon{:}Q] \qquad (\textit{Spec verification}) \end{array}}{D_{\mathtt{cs}}\ (\overline{\mathtt{t}}_2\ \overline{x})\ [\textit{static}\ \mathtt{sp}]\ \{\mathtt{S}\}}$$

To ensure sp meets the precondition for calling the superclass's constructor, we do an entailment checking for $P_c * \mathtt{P_f} \vdash P$: the precondition of the superclass P_c should entail the precondition of the subclass P with a possible anti-frame $\mathtt{P_f}$ that captures the extra nodes (do not appear in P_c) in the separation formula P. This anti-frame $\mathtt{P_f}$ is carried forward as part of the pre-states for verification. In addition, all extension fields of class D will be set to null before executing the constructor body S.

4.3 Proof Rules

This section presents primary proof rules specific to our OO language in Fig. 5. We leave the remaining standard rules [22,26] in Appendix C.

Rules Read, Write, NullRead and NullWrite are for object access (read-/write). Programmers typically check object type using instanceof before applying casting. Rules for instanceof, including InsNull, Ins and DyIns, model the type checking. While the first two are for objects with static views, the last one is for objects with dynamic views. C_m represents some classes with fields before C_i while C_k is for those after C_i. If $C_i \prec C$, intanceof operator returns true and drops all classes before C_i, but keeps the field information (of the dropped classes) in C_i. Otherwise, it returns false and drops those classes after C_i.

The rules for casting operators are CastNull, CastOk, CastErr, DyCastOk and DyCastErr. A casting error happens when the type of an object is assigned to an incompatible type. Note that the casting operation does not change the type stored in a heap or which method to call. A casting operator applies on a null value without any exceptions. Upcasting is always successful, as an instance of the subclass is also an instance of the superclass. Downcasting fails if we cast an object of exact type C to its subclass D. Casting to an unrelated class will also lead to an error. Similar to DyIns, rules DyCastOk and DyCastErr are for objects with dynamic view. If $C_i \prec C$, all classes after C_i in a dynamic view can be cast to C. Otherwise, all classes before C_i in a dynamic view can lead to casting errors.

Rules for method invocation are Static MethodInv and Dynamic MethodInv. When an object has an exact type C, we apply its static specification. For the dynamic method invocation ($\mathit{view}(x) = x :: ...D\langle ...\rangle$ means the dynamic view of object x in the precondition ends with type D), our system extracts reflexive specs according to the last type of the object's dynamic view. Note that, a reflexive spec may describe more classes than necessary. For instance, the dynamic view of an object x before a dynamic method invocation is $x{::}D\langle\overline{e_2}\rangle E\langle\overline{e_3}\rangle$. However, the dynamic view of *this* object in the precondition of the corresponding reflexive spec (class E) might be $\mathit{this}{::}C\langle\overline{e_1}\rangle D\langle\overline{e_2}\rangle E\langle\overline{e_3}\rangle$. It seems we could not apply the Dynamic MethodInv rule. In this case, we can use the Constancy rule

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

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

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

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

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

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

$$\frac{}{[x = \texttt{null} \wedge y = y']\ y{:=}(C)\ x\ [ok{:}\ x = \texttt{null} \wedge y = \texttt{null}]}\ \text{CastNull}$$

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

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

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

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

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

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

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

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

Fig. 5. Proof rules

in Appendix C and add a constrain $type(this) = \{D, E\}$ in the pre/post of the reflexive spec. Then, we can obtain a spec that can be used for this dynamic method invocation. Our case studies D utilise this strategy to do the proving. Constructor is for object constructor and is similar to Static MethodInv. Note that, all method invocations may need extra efforts for anti-frame inference. As the precondition P could contain more heap components than Pr for method calls, we need to infer a formula F where $Pr * F \vdash P$ and then push F forward by using the frame rule.

Theorem 1 (Soundness). *If* $\vdash [P]\mathtt{S}[\epsilon{:}Q]$, *then* $\vDash [P]\mathtt{S}[\epsilon{:}Q]$.

Proof. See Appendix B.

5 Implementation and Evaluation

Implementation. We prototype our incorrectness verification system for OOP, OURify, which consists of 10,000 lines of OCaml codes. We discharge the entailment checking and the anti-frame inference using the off-the-shelf tool, SLEEK [6,13].

OURify is an automated verifier that performs under-approximation compositional reasoning in a bottom-up manner. Specifically, given a program written in our core language (shown in Fig. 3) with well-annotated static and reflexive specifications, OURify verifies (i) the implementation against the static specifications; and (ii) behaviour subtyping conformance via the proposed subtyping relation among reflexive specifications. Afterwards, OURify reports the verification results, SUCCESS or FAILED, to the user.

OURify implements the proof rules in Fig. 5 and Appendix C, and a proof search algorithm. The algorithm takes a *specification table* T, that stores verified specifications of methods and uses a function $post(P, T, \mathtt{S})$, that computes the post-states ϵ': Q' of command $\mathtt{S}$ from its pre-condition P via applying the proof rules.

Given a method mn with the static specification $[P]\_[\epsilon{:}Q]$ and implementation m_c, OURify verifies the specification by first computing a set of post-states via $post(P, T, m_c)$. After that, for each post-state assertion $\epsilon' : Q'$, it invokes SLEEK to check whether ϵ' is the same with ϵ and $Q * emp \models Q'$. If there is no post-state that satisfies these checks, OURify returns FAILED. Otherwise, the static specification $[P]\_[\epsilon{:}Q]$ is verified. Theorem 1 ensures the correctness of the function *post*: $[P]\_[\epsilon{:}Q \mid \epsilon{:}\ Q \in post(P, T, m_c)]$. In addition, OURify checks the validity of the corresponding reflexive specification according to Definition 2 (specification subtyping between static and reflexive specifications of the method as well as reflexive specifications between methods of superclasses and subclasses), with the help of the back-end solver, SLEEK [6,13]. If all checks are successful, it returns SUCCESS. Otherwise, it produces FAILED.

Evaluation. The implementation is running on a Linux machine with an Intel i7 processor 3.40GHz and 8 GB of memory. We have tested OURify on programs with null-pointer-exceptions (NPE) and class-casting-exceptions (CAST) and reported the results in Table 1 while the name with "OK" indicates an ok program.

We have chosen 15 programs as our benchmarks. Six of them are manually constructed (those with the prefix M). The manually constructed programs are used to validate our implementation. The rest of the programs are taken from some existing works and publicly available data set[21,19,28,2,29]. Those programs have been translated into our core language. We annotate specifications for each method to capture their functional properties. The benchmarks are collections of commonly occurring bugs from various projects. We keep the crucial parts for doing the experiments. For instance, we have selected some benchmarks from Pulse repository [2]. The current version of Pulse does not support the detection of some OO-related bugs in those benchmarks. We are interested in those bugs in our system.

Table 1. Experimental results.

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

Table 1 summarises the experimental results. The table records: 1) LOC, the number of lines of code; 2) TIME, the running time (in seconds); 3) LoSpec, the number of lines of specifications – one pair of pre/post per line; 4) SUCCESS, the number of valid triples; and 5) FAILED, the number of invalid triples (all are false IL triples added to test OURify's soundness). The experimental results show that OURify verified all the triples correctly within a short running time and did not verify a false IL triple. Note that as our approach is compositional, the verification time increases linearly wrt. the number of specifications.

To demonstrate the practical impact of our proposal, we conduct the second experiment to reproduce the bugs reported by Pulse, an analyser developed within the Infer framework to find bugs in products at Meta [1]. Pulse applies under-approximate bi-abduction to infer static specifications automatically [12]. It reports a bug at a method only when it can derive a manifest er triple e.g., the triple is of the form $[emp \wedge true]\ code\ [er : q]$, where q is satisfiable.

For this experiment, we take all real-world programs in the above experiment. For each program, if Pulse reports an NPE bug, we construct corresponding IL triples, some of them are manifest er triples. If OURify could verify these triples, we classify the bug as confirmed. Otherwise, if we could not verify manifest er triples, we write either ok triples or latent er triples (which are er triples but not in the form of manifest) where OURify can verify them and classify the bug as unconfirmed. Moreover, we also carefully validated that the ones in confirmed partition are real bugs and all in unconfirmed one are false positives.

Table 2 presents the experimental results from both tools: 1) OK_OR, the number of ok specifications proved by OURify; 2) Cast_OR, the number of error specifications for casting errors proved by OURify; 3) NPE_OR, the number of error specifications for NPE proved by OURify; 4) Manifest, the number of manifest bugs (the true bugs, in contrast to latent/possible bugs [12]); 5)

NPE_PS, the number of NPE reported by Pulse; 6) Confirmed, the number of bugs reported by Pulse and confirmed by OURify; 7) FP_PS, the number of errors reported by Pulse but cannot be confirmed by OURify; and 8) FN_PS, the number of manifest bugs OURify could verify with er triples but Pulse did not discover.

Table 2. Incorrectness verification by OURify vs. bug finding by Pulse.

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

To sum up, there are 34 manifest bugs, including 16 confirmed bugs and 18 false negatives (missed by Pulse), and Pulse also reported 5 false positives. Interestingly, NPE_OR (which is 52) is higher than NPE_PS (which is 21) as NPE_OR includes specifications for both latent (may) and manifest (must) bugs while NPE_PS reports manifest bugs only. Furthermore, OURify can prove several manifest bugs which Pulse could not discover. (We discuss two case studies in Appendix D.) Most of these bugs relate to the hierarchical structure of OOP. For example, Pulse does not report bugs caused by the casting operator. In some situations, the superclass and the subclass behave differently (methods are overridden), as a result of which bugs are triggered when methods of the subclass are called but not the superclass. Pulse may miss such bugs. Requiring specification annotation is a drawback of OURify. It limits the applicability of OURify in large programs. However, writing specifications are always helpful to the program design. For instance, specifications can be used to support the regression analysis. Error specifications that are verified indicate the presence of bugs. They can kept to automatically remind programmers that certain errors should not re-appear (cannot be verified) when the code is modified in future. At the current stage, OURify works as a verification tool based on our proof system. We hope that our proof system could be the foundation for bug-finding tools, like Pulse, to hunt OO bugs more precisely in real codebases.

6 Related Work and Conclusion

Our work relates to the over-approximating verification for OOP [5,11,24,25]. To verify objects, Kassios [11] introduces a dynamic frame which describes data separation explicitly and could handle the aliasing problem. However, this work did not address behavioural subtyping which is essential for OOP. Parkinson and

Bierman [24] propose the *abstract predicate family* to handle behavioural subtyping in separation logic, including a mechanism to capture specifications where subclasses own more fields than their super-classes. Predicates inside a family can change the arity freely. Hence, the implication between formulae with different heap sizes can be proved through existentially quantified arguments. Later, two independent papers [5,25] propose the co-existence of static and dynamic specifications for OOP to uphold the Liskov substitution principle.

Following the landscape of the proposals in separation logic [5,24,25], we introduce the first proof system for under-approximating reasoning over OOP. Similar to the abstract predicate family, our dynamic view specifies the behaviours of multiple objects in a class inheritance relationship. In contrast, while the abstract predicate is a conjunction set (for over-approximation), the dynamic view is based on disjuncts (i.e., describing a set of objects for under-approximation) such that it could support `instanceof` and *casting* effectively. Furthermore, we use reflexive specifications to support dynamic dispatching in a modular manner (e.g., avoid re-verification) while the static specification provides a precise verification for static method calls.

Another essential concept in OOP is class invariant, which describes classes' functions [9]. Using class invariants helps to achieve more precise analyses in over-approximately verifying OOP. There are several challenging problems and solutions for around this concept. For example, Barnett *et al.* [3] propose a methodology that can reason about class invariants which could be temporarily broken while class fields are being updated. They use a special field to explicitly record if an object's invariant is valid. Leino and Müller [16] generalise ownership-based reasoning to support interrelated object invariants. An analogy of class invariant in IL is beyond this proposal and would be investigated in future.

Under-approximating reasoning in IL helps to avoid false positives which some static analysis tools are suffering [27]. Like IL, De Vries and Koutavas [30] proposed the reverse Hoare logic for under-approximation. Incorrectness separation logic (ISL) [26] enhances the applicability of IL in heap-manipulating programs. It combines separation logic [23] and IL, which provides the fundamental framework for our work. Le *et al.* [12] bring the ISL theory into practice. They developed Pulse-X to capture manifest bugs (bugs that will be triggered regardless of the calling context) in real-world projects. Our work, an IL logic for OOP, is meant to help build a foundational framework for under-approximating reasoning that could systematically support bug finding in OOP codebase.

Conclusion. This paper presents a variant of incorrectness separation logic to show the presence of bugs in Java-like OO programs. In particular, we introduce the static view and static specification to verify the implementation of a static method and the dynamic view and reflexive specification to verify behavioural subtyping. When behavioural subtyping holds, we can avoid costly case analysis for class objects. The reflexive specification can be further re-used for the dynamically dispatched method calls. For future work, we plan to extend the system with the bi-abduction technology to infer specs and automatically find bugs in real-world OO programs.

Acknowledgements The authors would like to thank anonymous reviewers for their comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier3 grant "Automated Program Repair", MOE-MOET32021-0001.

References

1. Infer static analyzer: Infer. `https://fbinfer.com/`. Accessed: 2023-06-02.
2. Pulse, an interprocedural memory safety analysis. `https://github.com/facebook/infer/tree/main/infer/tests/codetoanalyze/java/pulse`. Accessed: 2023-05-20.
3. Michael Barnett, Robert DeLine, Manuel Fähndrich, K Rustan M Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *J. Object Technol.*, 3(6):27–56, 2004.
4. Gavin M Bierman, MJ Parkinson, and AM Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.
5. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. *ACM SIGPLAN Notices*, 43(1):87–99, 2008.
6. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
7. William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 125–135, New York, NY, USA, 1989. Association for Computing Machinery.
8. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, page 258–267, USA, 1996. IEEE Computer Society.
9. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
10. Marieke Huisman and Bart Jacobs. Java program verification via a hoare logic with abrupt termination. In *International Conference on Fundamental Approaches to Software Engineering*, pages 284–303. Springer, 2000.
11. Ioannis T Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings 14*, pages 268–283. Springer, 2006.
12. Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.
13. Quang Loc Le, Jun Sun, and Shengchao Qin. Frame inference for inductive entailment proofs in separation logic. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–60, Cham, 2018. Springer International Publishing.
14. Gary T Leavens and David A Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):1–88, 2015.

15. Gary T Leavens and William E Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.
16. K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004.
17. Barbara Liskov. Keynote address-data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, 1987.
18. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, nov 1994.
19. Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.
20. Chenguang Luo and Shengchao Qin. Separation logic for multiple inheritance. *Electronic Notes in Theoretical Computer Science*, 212:27–40, 2008.
21. Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.
22. Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.
23. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.
24. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, 2005.
25. Matthew J Parkinson and Gavin M Bierman. Separation logic, abstraction and inheritance. *ACM SIGPLAN Notices*, 43(1):75–86, 2008.
26. Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing.
27. Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.
28. David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.
29. Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, pages 151–162, 2018.
30. Edsko de Vries and Vasileios Koutavas. Reverse hoare logic. In *International Conference on Software Engineering and Formal Methods*, pages 155–171. Springer, 2011.

A Semantics

Semantics of our core programs are defined in Fig. 6. Each method call will

$$
\begin{aligned}
&[\![\texttt{skip}]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s,h))\}\\
&[\![x{:=}e]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[x{\mapsto}s(e)],h))\}\\
&[\![x{:=}y.f]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[x{\mapsto}v],h))|\ h.2(s(y),f)=v\}\\
&[\![x{:=}y.f]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ s(y)=null\}\\
&[\![x.f{:=}y]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s,h'))|\ h'=(h.1,h.2[(s(x),f){\mapsto}s(y)])\}\\
&[\![x.f{:=}y]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ s(x)=null\}\\
&[\![y{:=}(C)\ x]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[y\mapsto s(x)],h))|\ (h.1(s(x))=C_1\wedge C_1{\prec}C)\vee s(x)=null\}\\
&[\![y{:=}(C)\ x]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ h.1(s(x))=C_1\wedge C_1{\not\prec}C\}\\
&[\![\texttt{S}_1;\texttt{S}_2]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|\epsilon=er\wedge((s,h),(s',h'))\in[\![\texttt{S}_1]\!]er\\
&\qquad\qquad\vee\exists(s'',h'').\ ((s,h),(s'',h''))\in[\![\texttt{S}_1]\!]ok\wedge((s'',h''),(s',h'))\in[\![\texttt{S}_2]\!]\epsilon\}\\
&[\![\texttt{t}\ x;\texttt{S}]\!]\epsilon \stackrel{\text{def}}{=} \{((s[x{\mapsto}v],h),(s'[x{\mapsto}v],h'))|((s,h),(s',h'))\in[\![\texttt{S}]\!]\epsilon\}\\
&[\![x.mn(\bar z)]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|(\exists s_1.\ ((s_o[\bar w{\mapsto}s(\bar z),this\mapsto s(x)],h),(s_1,h'))\in[\![\texttt{S}]\!]ok\\
&\qquad\qquad\wedge((s_1,h'),(s'=s[ret{\mapsto}s_1(y)],h'))\in[\![\texttt{return}\ y]\!]ok)\\
&\qquad\vee(\exists\texttt{S}',s_1.\epsilon=er\wedge((s_o[\bar w{\mapsto}s(\bar z),this\mapsto s(x)],h),(s_1,h'))\in[\![\texttt{S}']\!]er\wedge s'=s)\\
&\qquad\vee((s',h')=(s,h)\wedge\epsilon=er\wedge s(x)=null)\}\\
&\qquad\qquad\text{provided that } h.1(s(x))=C,\ body(C.mn(\bar w))=\{\texttt{S};\texttt{return}\ y\};\\
&\qquad\qquad s_o \text{ is new method stack;}\\
&\qquad\qquad \texttt{S}' \text{ is a sub-sequence of statements (from beginning) of } \texttt{S}\\
&[\![\texttt{return}\ y]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s',h))|\exists s''.\ s'=s''[ret{\mapsto}s(y)]\}\\
&[\![\texttt{new}\ C(\bar y)]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|\ \exists l.\ loc(l)\notin dom(h.1)\\
&\qquad\qquad\wedge l.1(loc(l))=C\wedge l.2[\overline{(loc(l),f)}{\mapsto}null]\wedge\\
&\qquad\qquad((\exists s_1.\ ((s_o[\bar w{\mapsto}s(\bar y)],h\uplus l),(s_1,h'))\in[\![\texttt{S}]\!]ok\\
&\qquad\qquad\qquad\wedge((s_1,h'),(s'=s[ret{\mapsto}loc(l)],h'))\in[\![\texttt{return}\ loc(l)]\!]ok)\\
&\qquad\qquad\vee(\exists\texttt{S}',s_1.\epsilon=er\wedge((s_o[\bar w{\mapsto}s(\bar y)],h\uplus l),(s_1,h'))\in[\![\texttt{S}']\!]er)\wedge s'=s))\}\\
&\qquad\qquad\text{provided that } body(C(\bar w))=\{\texttt{S}\},\ \texttt{S}' \text{ is a sub-sequence of } \texttt{S};\\
&\qquad\qquad loc(l) \text{ returns the location of } l;\ s_o \text{ is new method stack}
\end{aligned}
$$

Fig. 6. Semantics of the core OO language.

create a new stack s_0 (called method scope) [4] and store the mapping of the input parameters. A `return` statement will remove the method scope and return the value to the original stack. The statement `new` $C(\bar y)$ instantiates a new object

$$[\![emp]\!] \stackrel{\text{def}}{=} \{(s,h)|\ dom(h.1) = \emptyset \wedge dom(h.2) = \emptyset\}$$

$$[\![x.f{\mapsto}e]\!] \stackrel{\text{def}}{=} \{(s,h)|\ h.2(s(x),f) = s(e) \wedge dom(h.2) = \{(s(x),f)\}\}$$

$$[\![x : C]\!] \stackrel{\text{def}}{=} \{(s,h)|\ h.1(s(x)) = C \wedge dom(h.1) = \{s(x)\}\}$$

$$[\![x{\mapsto}C\langle\overline{e}\rangle]\!] \stackrel{\text{def}}{=} \{(s,h)|\ h.1(s(x)) = C * (\underset{f_i \in field(C)}{*} h.2(s(x),f_i) = s(e_i))$$
$$\wedge\ dom(h.1) = \{s(x)\} \wedge dom(h.2) = \{\overline{(s(x),f)}\}\}$$

$$[\![x{::}C\langle\overline{e}\rangle]\!] \stackrel{\text{def}}{=} [\![x{\mapsto}C\langle\overline{e}\rangle]\!]$$

$$[\![x{::}C_1\langle\overline{e_1}\rangle{\cdots}C_{n-1}\langle\overline{e_{n-1}}\rangle C_n\langle\overline{e_n}\rangle]\!] \stackrel{\text{def}}{=} [\![x{\mapsto}C_n\langle\overline{e_1},\cdots,\overline{e_n}\rangle]\!] \vee [\![x{::}C_1\langle\overline{e_1}\rangle\cdots C_{n-1}\langle\overline{e_{n-1}}\rangle]\!]$$

$$[\![\kappa_1 * \kappa_2]\!] \stackrel{\text{def}}{=} \{(s,h)|\ \exists h', h''.\ h.1 = h'.1 \bullet h''.1 \wedge h.2 = h'.2 \bullet h''.2$$
$$\wedge\ (s,h') \in [\![\kappa_1]\!] \wedge (s,h'') \in [\![\kappa_2]\!]\}$$
$$\texttt{where}\ h'.i \bullet h''.i \stackrel{\text{def}}{=} \begin{cases} h'.i \uplus h''.i & \texttt{if}\ dom(h'.i) \cap dom(h''.i) = \emptyset \\ undefined & \texttt{otherwise} \end{cases}$$

Fig. 7. Semantics of the assertion language.

on the heap. The constructor is a unique method. At the initial step, a heap l will be allocated to this object and set all its fields to *null*. Then, the statements in the body will be executed like a standard method and implicitly return the location of l at the end. We assume a reserved variable *ret*, which captures the value in a return statement and will be replaced once assigned. For `void` method, it returns *nothing*. The conditional statement can be encoded by `assume` and choice `S+S` [26]. For brevity, we omit the details.

In IL, the abnormal states are captured explicitly. For example, there are three scenarios causing null-pointer-exceptions including reading or updating a field from *null*, cf. $[\![x{:=}y.f]\!]er$ and $[\![x.f{:=}y]\!]er$ respectively. The third case happens in the static method call, i.e., when the receiver of a method is *null*.

The semantics of assertions are defined in Fig. 7.

B Soundness

Lemma 2 (soundness of reflexive specification). *Suppose we have a class hierarchy $C_1, C_2...C_n$ and all the static specifications are sound for the method mn (no dynamically dispatched method call in the body), i.e. $\vDash [S_{pr_i}]\ C_i.mn\ [\epsilon{:}S_{po_i}]$. Then, the reflexive spec of mn in C_n is sound, that is $\vDash [D_{pr_n}]\ x.mn\ [\epsilon{:}D_{po_n}]$ where x can be one of the type in $C_1, C_2...C_n$.*

Proof. Suppose the hierarchy only has one class C_1, the static specification is just a subtype specification of the corresponding reflexive specification. By the consequence rule of IL, we have $\vDash [D_{pr_1}]\ x.mn\ [\epsilon{:}D_{po_1}]$. Suppose the hierarchy has i classes $C_1, C_2...C_i$ and $\vDash [D_{pr_{i-1}}]\ x.mn\ [\epsilon{:}D_{po_{i-1}}]$ where the possible types of x exclude C_i. We have the static specification $[S_{pr_i}]\_[\epsilon{:}S_{po_i}]$ and the reflexive specification $[D_{pr_i}]\_[\epsilon{:}D_{po_i}]$ for C_i. According to the requirement of Definition 2, we

have $[S_{pr_i}]\models[D_{pr_i}]$ and $[D_{po_i} \wedge type(this) \in \{C_i\}]\models[S_{po_i}]$. By consequence rule, we have $\models [D_{pr_i}]\ x.mn\ [D_{po_i} \wedge type(this) \in \{C_i\}]$ when the type of x is only C_i. Similarly, we require $[D_{pr_{i-1}}]$-$[\epsilon{:}D_{po_{i-1}}]$ $<:_{C_{i-1}}$ $[D_{pr_i}]$-$[\epsilon{:}D_{po_i}]$. Then, we have $\models [D_{pr_i}]\ x.mn\ [D_{po_i} \wedge type(this) \in \{C_{i-1}\}]$ (the type of x is not C_i).

Therefore, by induction, $\models [D_{pr_n}]\ x.mn\ [\epsilon{:}D_{po_n}]$.

Lemma 3 (soundness of axioms). *The axioms are true, i.e. they are valid under the relational denotational semantics.*

Proof. Some of the axioms are analogous and follow the same proving idea as [26]. We shall not repeat them. However, we should focus on *instanceof* and casting which are not mentioned anywhere else.

– $x\ instanceof\ C$:
InsNull:
Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [ok{:}\ x = null \wedge y = false]$, such that $h\#h_{po}$ (# means disjoint). By definition, we know that $s_{po}(x) = null$, $s_{po}(y) = false$ and $dom(h_{po}) = \emptyset$. Let $s_{pr} = s_{po}[y{\mapsto}y']$ for any y' and $h_{pr} = h_{po}$. By definition $(s_{pr}, h_{pr}) \in [x = null]$. Then, it suffices to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{pr})) \in [\![y{:=}x\ instanceof\ C]\!]ok$.

Ins1:
Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [ok{:}\ x : C_1 \wedge y = True \wedge C_1 \prec C]$ such that $h\#h_{po}$. By definition, $s_{po}(y) = true$ and $h_{po}.1(s_{po}(x)) = C_1$. Let $s_{pr} = s_{po}[y \mapsto y']$ for any y' and $h_{pr} = h_{po}$. By definition, we know $(s_{pr}, h_{pr}) \in [x : C_1 \wedge y = y' \wedge C_1 \prec C]$. Since $h(s_{po}(x)).1 = h(s_{pr}(x)).1 = C_1$. it suffice to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{po})) \in [\![y{:=}x\ instanceof\ C]\!]ok$.

Ins2:
Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [ok{:}\ x : C_1 \wedge y = false \wedge C_1 \not\prec C]$ such that $h\#h_{po}$. By definition, $s_{po}(y) = false$ and $h_{po}.1(s_{po}(x)) = C_1$. Let $s_{pr} = s_{po}[y \mapsto y']$ for any y' and $h_{pr} = h_{po}$. By definition, we know $(s_{pr}, h_{pr}) \in [x : C_1 \wedge y = y' \wedge C_1 \not\prec C]$. It suffice to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{po})) \in [\![y{:=}x\ instanceof\ C]\!]ok$.

DyIns: When the postcondition is q_1, it follows Ins1. When the postcondition is q_2, it follows Ins2.

– $y{:=}C\ (x)$:
CastNull:
Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [ok{:}\ x = null \wedge y = null]$, such that $h\#h_{po}$ (# means disjoint). By definition, we know that $s_{po}(x) = null$, $s_{po}(y) = null$ and $dom(h_{po}) = \emptyset$. Let $s_{pr} = s_{po}[y{\mapsto}y']$ for any y' and $h_{pr} = h_{po}$. By definition $(s_{pr}, h_{pr}) \in [x = null \wedge y = y']$. Then, it suffices to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{po})) \in [\![y{:=}\ (C)\ x]\!]ok$.

CastOk:

Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [ok{:}\ x{\mapsto}C_1\langle\bar{e}\rangle[y'/y] \wedge x = y \wedge C_1{\prec}C]$ such that $h\#h_{po}$. By definition, $s_{po}(x) = l$,$s_{po}(x) = s_{po}(y)$, $h_{po}.1(l) = C_1$ and $h_{po}.2(\overline{l, f}) = \bar{e}[y'/y]$. Let $s_{pr} = s_{po}[y \mapsto s_{po}(y')]$ and $h_{pr} = h_{po}$. By definition, we know $(s_{pr}, h_{pr}) \in [x{\mapsto}C_1\langle\bar{e}\rangle \wedge y = y' \wedge C_1{\prec}C]$. It suffice to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{po})) \in [\![(C_2)\ x]\!]ok$.

CastErr:
Pick an arbitrary heap h and $(s_{po}, h_{po}) \in [er{:}\ x : C_1 \wedge C_1{\not\prec}C]$ such that $h\#h_{po}$. By definition, $s_{po}(x) = l$, $h_{po}.1(l) = C_1$. Let $s_{pr} = s_{po}$ and $h_{pr} = h_{po}$. By definition, we know $(s_{pr}, h_{pr}) \in [x : C_1 \wedge C_1{\not\prec}C]$. It suffice to show $((s_{pr}, h \uplus h_{pr}), (s_{po}, h \uplus h_{po})) \in [\![y := (C)\ x]\!]err$.

DyCastOk: it follows CastOk.

DyCastErr: it follows CastErr.

– Method call:
Null MethodInv: It directly follows the semantics.

Other rules for method call: The primary axioms can verify the static specification of methods. By Lemma 2, the corresponding reflexive specifications are also valid. Hence, our rules Static MethodInv, Dynamic MethodInv and Constructor are sound (by consequence rule).

C Other proof rules

The rules Skip, Assign and Assume are standard. The rule Choice states that we can drop one of the branches. For the sequence statement $S_1; S_2$ states that if S_1 leads to error states, we can skip the remaining statements. Local handles local variables with existentially quantified variables.

$$\text{Skip}\quad [emp]\ skip\ [ok{:}\ emp] \qquad \text{Assign}\quad [x{=}x']\ x{:=}e\ [ok{:}\ x{=}e[x'/x]] \qquad \text{Assume}\quad [emp]\ assume(B)\ [ok{:}\ B]$$

$$\frac{[p]\ \mathtt{S_i}\ [\epsilon{:}q] \qquad i \in \{1,2\}}{[p]\ \mathtt{S_1 + S_2}\ [\epsilon{:}q]}\ \text{Choice} \qquad \frac{[p_1]\ \mathtt{S}\ [\epsilon{:}q_1] \quad [p_2]\ \mathtt{S}\ [\epsilon{:}q_2]}{[p_1 \vee p_2]\ \mathtt{S}\ [\epsilon{:}q_1 \vee q_2]}\ \text{Disj}$$

$$\text{Seq1}\ \frac{[p]\ \mathtt{S_1}\ [er{:}\ q]}{[p]\ \mathtt{S_1; S_2}\ [er{:}\ q]} \qquad \text{Seq2}\ \frac{[p]\ \mathtt{S_1}\ [ok{:}\ r] \quad [r]\ \mathtt{S_2}\ [\epsilon{:}q]}{[p]\ \mathtt{S_1; S_2}\ [\epsilon{:}q]} \qquad \text{Local}\ \frac{[p]\ \mathtt{S}\ [\epsilon{:}q]}{[p]\ \mathtt{t}\ x; \mathtt{S}\ [\epsilon{:}\exists x.q]}$$

Note that the consequence rule can be applied when the precondition is weakened and the postcondition is strengthened, which is a reversed version of the consequence rule in over-approximation verification. ISL [26] introduces the negative heap $x \not\mapsto$ to denote a deallocated location. This introduction helps to retain the soundness of the frame rule for under-approximating analysis. As our programming language does not contain `delete` statement (i.e., we do not allow to explicitly remove objects from the heap), the proposed assertion language does

not include the negative heap. Alternatively, we use *null* to mark invalidated heaps (e.g., uninitialised objects).

$$\text{Constancy}\quad \frac{[p]\ S\ [\epsilon{:}q] \qquad mod(S)\cap fv(r){=}\emptyset}{[p\wedge r]\ S\ [\epsilon{:}q\wedge r]}$$

$$\text{Consequence}\quad \frac{p'{\Rightarrow}p \qquad [p']\ \mathtt{S}\ [\epsilon{:}q'] \qquad q{\Rightarrow}q'}{[p]\ \mathtt{S}\ [\epsilon{:}q]} \qquad \text{Frame}\quad \frac{[p]\ \mathtt{S}\ [\epsilon{:}q] \qquad mod(\mathtt{S})\cap fv(r){=}\emptyset}{[p*r]\ \mathtt{S}\ [\epsilon{:}q*r]}$$

D Case studies

```
1  class AnInterface {...}
2  class ImpleOne extends AnInterface {...}
3  class ImpleTwo extends AnInterface {...}
4  ...
5  virtual int somevalue(AnInterface i)
       static  [i :: AnInterface⟨⟩ImpleOne⟨⟩]_
6      [ok: ∃impl. i ↦ ImpleOne⟨⟩∧impl = i ∧ ret = ...][er: i ↦ AnInterface⟨⟩]
       static  [i :: AnInterface⟨⟩ImpleTwo⟨⟩]_[er: i :: AnInterface⟨⟩ImpleTwo⟨⟩]
       {ImpleOne impl := (ImpleOne) i;
7                  return impl.getInt();}
8
9  virtual int classCastException()
10     static  [true]_[er: a ↦ ImpleTwo⟨⟩]
11     {ImpleTwo a := new ImpleTwo();
12     return somevalue(a);}
13 ...
```

Fig. 8. Another casting error

Case Study 1: Figure 8 shows a program (simplified) taken from [2]. Two non-related subclasses `ImpleOne` and `ImpleTwo` extend the common superclass `AnInterface` respectively. A method `somevalue` takes an object i of static type `AnInterface` as an argument. The method casts i into a subclass `ImpleOne` and returns a value by calling its `getInt`() method. There is a latent bug in this method as the casting will be successful if the actual type of i is `ImpleOne`; otherwise, there will be casting errors. We have two specifications to capture this scenario (reflexive specifications are omitted):

- i has the dynamic view of `AnInterface`⟨⟩`ImpleOne`⟨⟩. The DyCastOk and DyCastErr will split them into two cases for casting. If $i \mapsto$ `ImpleOne`⟨⟩, the program will successfully return some value (omitted in the specification). If $i \mapsto$ `AnInterface`⟨⟩, the program enters an abnormal execution.
- i has the dynamic view of `AnInterface`⟨⟩`ImpleTwo`⟨⟩. The DyCastErr directly concludes that the program enters an abnormal execution after casting.

The next method `classCastException` has a manifest bug, i.e. regardless of the calling context, the bug will be triggered. The method body instantiates object a with an allocated type `ImpleTwo`. Subsequently, call `somevalue(a)`. As the type of i is not modified in `somevalue`, we can use Constancy rule to extract a suitable specification for this call.

$$\frac{[i :: \texttt{AnInterface}\langle\rangle\texttt{ImpleTwo}\langle\rangle]\_[er\text{: } i :: \texttt{AnInterface}\langle\rangle\texttt{ImpleTwo}\langle\rangle]}{[i :: \texttt{AnInterface}\langle\rangle\texttt{ImpleTwo}\langle\rangle \wedge type(i) = \texttt{ImpleTwo}] \quad - \quad [er\text{: } i :: \texttt{AnInterface}\langle\rangle\texttt{ImpleTwo}\langle\rangle \wedge type(i) = \texttt{ImpleTwo}]}{[i \mapsto \texttt{ImpleTwo}\langle\rangle]\_[er\text{: } i \mapsto \texttt{ImpleTwo}\langle\rangle]}$$

By applying Static MethodInv, we can verify the specification of this method.

Case Study 2: In this case study, we test OURify over a program in which the subclass is not behavioural subtyping. Fig. 9 shows an example of the radial subclass.

```
1  class Super {
2      virtual Object foo()
3      static [this↦Super⟨⟩]_[ok: ∃o. this↦Super⟨⟩ * o↦Object⟨⟩ ∧ ret=o]
4      reflex [this::Super⟨⟩]_[ok: ∃o. this::Super⟨⟩ * o↦Object⟨⟩ ∧ ret=o]
5      {Object o := new Object(); return o;}}
6
7  class Sub extends Super {
8      override Object foo()
9      static [this↦Sub⟨⟩]_[ok: this↦Sub⟨⟩ ∧ ret=null]
10     reflex [this::Super⟨⟩Sub⟨⟩]_
           [ok: ∃o. this::Super⟨⟩ * o↦Object⟨⟩ ∧ ret=o][ok: this::Sub⟨⟩ ∧ ret = null]
11     {return null;}}
12 ...
13 virtual void test(Super a)
14     static [a::Super⟨⟩Sub⟨⟩]_
15             [ok: ∃m, o. a::Super⟨⟩ * o↦Object⟨⟩ ∧ m = o]
16             [er: ∃m. a::Sub⟨⟩ ∧ m=null]
17     {Object m := a.foo(); m.toString();}
18
19 virtual void buggy(Sub b)
20     static [b::Sub⟨⟩]_[er: b::Sub⟨⟩]
21     {test(b);}
22 ...
```

Fig. 9. Non-behavioural subclass

The method `foo()` is overridden in `Sub` and it is radically different to the one in `Super`: It returns `null`, while `Super.foo()` returns an `Object`. The `test(Super a)` method makes a dynamic dispatching call of `foo` and the returned object will be a caller of `toString`. As `Sub.foo` returns *null*, if the actual type of `a` is `Sub`, *null*.`toString` leads an NPE while there will be no error if the actual type of `a` is

`Super`. The `buggy(Sub b)` calls method `test` on a `Sub` object which will lead to a manifest bug. We note that showing the presence of the bug in this example is non-trivial. For instance, the program is tested by Pulse but Pulse could not detect the manifest bug in `buggy`.

For reflexive specifications, we use a dynamic view to specify `Sub.foo()`'s functions. One describes its own function, and the other one is for its superclass. To prove the implementation against this specification, OURify generates the following three proof obligations: i) $\texttt{static}(\texttt{Super.foo}()) <:_U \texttt{reflex}(\texttt{Super.foo}())$; ii) $\texttt{static}(\texttt{Sub.foo}()) <:_U \texttt{reflex}(\texttt{Sub.foo}())$;
and iii) $\texttt{reflex}(\texttt{Super.foo}()) <:_U \texttt{reflex}(\texttt{Sub.foo}())$ after the verification of static specifications. The first obligation is straightforward. For the second one, the pre-conditions checking $[this{\mapsto}\texttt{Sub}\langle\rangle]{\models}[this{::}\texttt{Super}\langle\rangle\texttt{Sub}\langle\rangle]$ is also trivial. For post-conditions, after conjoining with $type(this) \in \{\texttt{Sub}\}$, the first ok postcondition becomes false and the second ok postcondition trivially implies the postcondition of the static spec. Similarly, the third proof obligation can be proven by conjoining $type(this) \in \{\texttt{Super}\}$ with the postcondition of $\texttt{reflex}(\texttt{Sub.foo}())$.

Now, let us consider the methods `test` and `buggy`. We omit the reflexive specification of the two methods as they are virtual.

The augment `a` of `test` has static type `Super`. Then, the method call `foo()` is dynamically dispatched as the actual type of `a` can be either `Super` or `Sub`. Hence, OURify re-uses the reflexive specification for `foo()` in `Sub`. By applying our Dynamic MethodInv rule, we reach two possible intermediate states:

$$[ok\colon \exists o.\ a{::}\texttt{Super}\langle\rangle * o{\mapsto}\texttt{Object}\langle\rangle \wedge m = o] \tag{1}$$

$$[ok\colon a{::}\texttt{Sub}\langle\rangle \wedge m{=}null] \tag{2}$$

OURify analyses the two states separately. For state (1), this method will terminate normally (for simplicity, we assume `toString()` method has no effect on states). However, in the state (2), the method call `toString`() leads to an error post-state as $m = null$ (Null MethodInv rule).

The `buggy` method has a manifest bug, because it runs the method `test` only with a `Sub` object. To verify this method, OURify re-uses the specification of `test`. Similar to case study one, by applying Constancy rule, OURify can conjoin $type(a) = \texttt{Sub}$ (as the type a is not modified) with the pre/post of `test` to extract the following specification:

$$[a{::}\texttt{Sub}\langle\rangle]\_[er\colon \exists m.\ a{::}\texttt{Sub}\langle\rangle \wedge m{=}null]$$

This specification is subsequently used for the method call $\texttt{test}(b)$. By applying the Static MethodInv rule and Consequence rule, OURify verifies the er specification of `buggy`.