Inferring Incorrectness Specifications for Object-Oriented Programs

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

[1] National University of Singapore, Singapore
[2] University College London, United Kingdom

Abstract. Incorrectness logic (IL) based on under-approximation is effective at finding real program bugs. The prior work utilises bi-abductive specification inference mechanism to infer IL specifications for analysing large-scale C projects. However, this approach does not work well with object-oriented (OO) programs because it does not account for *class inheritance* and *method overriding*. In our work, we present an IL specification inference system that tackles these issues. At its core, we encode type information in our bi-abductive reasoning and propagate *type constraints* throughout the analysis. The direct benefit is that we can efficiently identify bugs caused by improper usage of the casting operator, which cannot be handled by the existing specification inference. Meanwhile, our system can reduce false positives while finding more true bugs because of not losing OO-type information. Furthermore, we model dynamic dispatching calls by inferring *dynamic specifications*, where the possible types of the calling object at runtime are bounded by the *type constraints*. We prototype our system in ToolX and evaluate it using real-world projects. Experimental results show that it finds 400% more class-cast-exceptions compared with Error Prone and improves the precision of finding null-pointer-exceptions by 27.0% compared with Pulse.

1 Introduction

Incorrectness Logic (IL) [31], as a dual to Hoare logic (HL), is an effective and principled approach for proving the presence of bugs. A recent work [21] implements a tool called Pulse-X to infer IL specifications within the Meta/Infer framework and aims at real bug detection for large C-based projects. In Pulse-X, IL with its extension via Incorrectness Separation Logic (ISL) are used together with bi-abduction [13] to infer specifications automatically. Pulse-X has been shown to be effective in finding bugs in real-world projects such as OpenSSL.

The current IL bi-abductive inference mechanism [21] only associates every variable with its declared type during the analysis. However, this is inadequate for modelling OO programs. In OO programming, types form class hierarchies and declared types encompass themselves and all their subclasses. Consequently, a method could accept multiple types during the real execution. These characteristics create challenges for the existing inference mechanism.

Firstly, it cannot handle the casting operation, which is widely used in OO programs. The casting operation is in the form of $(C)\,e$, which casts the source

type of the value by evaluating expression e to type C. Casting operations can cause system failures, i.e., class-cast-exception (CCE), at run-time when the source type is not a subtype of C. Research [15,20,25,30] has shown that the CCE stands as one of the most pervasive bugs in OO programs. Unfortunately, the current approach could not analyze casting operations as it can not recognize object type possibilities. In addition, a lack of OO-type information leads to false positives, as some bugs will only occur if some type constraints are satisfied. However, as variables are type-insensitive in [21], it may report infeasible bugs.

Furthermore, because this mechanism uses fixed types, each method call, e.g., $x.mn()$, is considered to be statically dispatched. Then, the analysis could be imprecise. Suppose the type declared for x is an interface; it could not find a specification as $mn()$ does not have an implementation in the interface. If the declared type of x is a normal class, it loses precision due to the ignorance of subtypes and method overriding in OO programs. Considering these unsolved issues are crucial in OO programs, the current inference approach must be advanced.

Incorrectness Logic and Bi-abduction. $[P]\ S\ [\epsilon{:}Q]$ denotes an IL triple. Here, $\epsilon \in \{ok, er\}$ captures symbolic traces of successful or error outcomes. Intuitively, an IL triple is valid if every program state satisfying the postcondition is reachable from some program states satisfying the precondition. A key feature of IL is that it allows dropping execution paths while ensuring all described paths are true in actual executions. Hence, an error postcondition $[er{:}\ ...]$ stands for true bugs. A bi-abduction problem $P * M \vdash_{bi} Q * F$ is to abduce a missing formula M, which is necessary to execute a command and calculate an unchanged *frame* F. Bi-abductive reasoning can generate HL specifications automatically. As IL is dual to HL, Pulse-X adapts bi-abduction to infer IL specifications due to the flipped consequence rule. Specifically, the IL bi-abduction problem is $Q * F \vdash_{bi} P * M$ where M is inferred via frame calculation. Pulse-X analyses each method starting from an $emp \wedge true$ formula, while in our system, the initial condition will record all declared type information. Our system builds up and propagates *type constraints* throughout the reasoning, accommodating the bug finding for CCEs and recording the possible types for dynamic dispatching in real executions.

Errors in OO Programs and Error Reporting. In this work, we target CCEs and null-pointer-exceptions (NPEs), which occur when trying to access a null pointer that does not point to an object. For NPEs, not all the *possible errors* are of programmers' interest. For example, the method $foo(A\ a)\{a.mn()\}$ can trigger an NPE when null is given as its input. The programmer may reason that a will rarely be null and decide to ignore this possible NPE.

To systematically decide if an error is worth reporting and reduce false positives, Le et al. [21] defined *manifest bugs*, which persistently occur regardless of the input values, and *latent bugs* which only occur for some input values. Following the convention, in this work, we also target manifest bugs for NPEs. Pulse-X may generate multiple specifications for one analyzed method. Each specification is associated with one path of the program. However, to determine manifest bugs, they examine specifications individually while ignoring the bugs that exist

in multiple paths. We propose a *merging* mechanism which generalises the reporting strategies to discover more true bugs. In addition, dynamic dispatching calls introduce a large set of paths, as each possible type leads to a different set of paths, which worsens the path explosion problem. The proposed *merging* mechanism can mitigate the problem by combining compatible specifications. The mechanism reduces the path space without sacrificing path information. On the other hand, we argue that latent CCEs are also worth reporting. For example, the programmers may not be aware of the entire class hierarchy and ignore some type possibilities for input objects. Some inputs are fine, but those ignored objects could be dangerous, especially when the code is re-used or used externally. Hence, we further relax the reporting criteria which covers a larger set of interesting bugs. Our contributions are:

- We propose an IL specification inference system for OO programs. Our system is type-sensitive, such that it can effectively reason about OO features and find bugs which cannot be handled by the existing inference system.
- We propose bug reporting criterion for both NPEs and CCEs. The NPEs reporting criteria is a generalisation of the existing work via *merging* and the *merging* mechanism can also mitigate the path explosion issue.
- We implement the inference mechanism in a tool called ToolX. Our experimental results show that our tool outperforms the state-of-the-art tools. The source code of the ToolX is available from [7].

2 Motivating Examples

Our motivation examples demonstrate that our approach can effectively detect CCEs, and increase the precision of the existing static analysis for OO programs.

2.1 Detecting Class-Cast-Exceptions

Fig. 1 shows a *possible* casting error found by ToolX. The input of this method is a *COSBase* object. In the if branch, the developer uses an *instanceof* operator to guard the casting (*COSObject*) *o*. However, in the else branch, the developer directly casts the object *o* to *COSDictionary*, which may cause a run-time exception as there exist classes that are neither subtypes of *COSObject* nor *COSDictionary*. This issue has been existing for more than ten years, and fixed by the developer recently (June, 2024). To identify this bug, ToolX starts with an initial program state, $\phi_0 = (ty(o) \prec: COSBase)$, meaning that the input type of *o* is *COSBase* or *COSBase*'s subclasses. At line 4, ToolX extends the state with

```
1 private COSDictionary toDictionary(COSBase o){
2   if (o instanceof COSObject){
3       return (COSDictionary)((COSObject)o).getObject();}
4   else{return (COSDictionary)o;}} //may cause a run-time error
```

Fig. 1. A Casting Error Found in an Open-Source Project Pdfbox [6]

the type constraint: $\phi_1 = (ty(o) \not<: COSObject)$. When analysing line 4, ToolX explores the possibility of CCE, which is when $\phi_2 = (ty(o) \not<: COSDictionary)$. Since ϕ_2 does not contradict to the current state, ToolX infers an error specification with a precondition containing $\phi_0 \wedge \phi_1 \wedge \phi_2$.

2.2 Increasing Bug-finding Precision

ToolX is more accurate than the state-of-the-art tool Pulse, the commercial version of Pulse-X, by reducing false positives while finding more true positives.

Reduce False Positives. As shown in Fig. 2, Pulse reports a bug at line 2, which calls the method defined at line 4 by passing *null* as the second argument. As the second formal argument, object *icon* is dereferenced at line 7 to access its method *getImage*(); and if *icon* is *null*, there is an NPE. Hence, Pulse reports this error.

However, as this method call is under an *instanceof* checking and *null* is not an instance of any class, *icon*'s value will never be *null* at line 7. Therefore, there is no NPE. ToolX avoids such false positives by inferring specifications containing precise type constraints. The precondition inferred for entering the if branch at line 7 contains a condition $ty(icon) \prec: ImageIcon$. Then, ToolX finds that the method call at line 2 does not take this precondition as a valid call since $icon = null$. Thus, ToolX does not report any NPEs.

```
1 public ErrorDialog(JComponent owner, Throwable t){
2   this(owner, null, t); } // this is a false positive NPE
3
4 public ErrorDialog(JComponent owner, Icon icon, Throwable t){
5   ...
6   if (icon instanceof ImageIcon){
7       setIconImage(((ImageIcon) icon).getImage());}
8   else {...}}
```

Fig. 2. A (Simplified) False Positive Reported by Pulse [4]

Find True Positives. Fig. 3 presents a buggy program from Infer's test repository [18]. Unfortunately, this bug has existed in this repository for several years but still cannot be found by its toolchain. There are two classes declared in this example where B is a subclass of A. B overrides the method *foo*() such that A.*foo*() returns a new *Object* instance, while B.*foo*() returns *null*. The method *dyn_mn* takes the object o as the input, and o could be either an instance of A or an instance of B. The method executes normally if it has type A but throws an NPE if it has type B. Pulse could not detect such bugs as it only analyses

```
1 class A {Object foo() {return new Object();}}
2 class B extends A { @Override Object foo() {return null;}}
3 void dyn_mn(A o) {o.foo().toString();}
4 void buggy(B b) {dyn_mn(b);}  // this is a true bug
```

Fig. 3. A True NPE in Infer's Test Repository

the case where o is type A and fails to consider the other possible type B. In addition, this bug becomes manifest in method *buggy* at line 4 as it calls method *dyn_mn* by always passing a B type instance as an input. However, Pulse does not support the reasoning for the dynamic dispatching call shown in the example, i.e., it ignores the overriding method in the subclass. As such, it could not derive the error specification for *buggy*. This example highlights the need for a systematic method to catch and report such bugs in OO programs.

In our approach, ToolX infers the *static specifications* for both $A.foo()$ and $B.foo()$ according to their implementations, respectively. Meanwhile, ToolX composes a *dynamic specification* for $A:foo()$ from the earlier inferred static specifications of both A and B. The notation $A:foo()$ means that *foo* is dynamically dispatched. ToolX utilises the dynamic specification to cover the behaviour when o is type B in *dyn_mn* and captures the missing bug in *buggy*.

3 Target Language and Specification Language

$$
\begin{array}{rcl}
\mathscr{P} & ::= & \overline{cdef};\\
\tau & ::= & int \mid bool \mid void\\
meth & ::= & t\ mn\ (\overline{t}\ \overline{x})\ \{S\}\\
S & ::= & skip \mid e \mid t\ x; S \mid S; S \mid S + S \mid S^*\\
e & ::= & v \mid x{:=}e \mid y := x.f \mid x.f{:=}y \mid error() \mid new\ C(\overline{x}) \mid\\
 & & x.mn(\overline{y}) \mid x\ instanceof\ C \mid (C)\ x \mid assume(b)
\end{array}
\qquad
\begin{array}{rcl}
cdef & ::= & class\ C_1\ extends\ C_2\ \{\overline{t}\ \overline{f};\ \overline{meth}\}\\
t & ::= & C \mid \tau\\
v & ::= & const \mid x
\end{array}
$$

Fig. 4. A Core Object-Oriented Language

Fig. 4 presents our target OO language, which is call by value and uses single inheritance. The entire class hierarchies of a program are constructed via *extends* keyword. *Object* is an implicit superclass of all classes; $x, y...$ range over variables. The *const* represents the constant values. Following the encoding convention [31,21], we represent conditionals as $(assume(b); S_1)+(assume(\neg b); S_2)$ where b is a Boolean value and $+$ is a non-deterministic choice between two statements; and *while* is encoded as $(assume(b); S)^*; assume(\neg b)$ where $*$ is the Kleene star iteration. The semantics of the core language can be found in the Appendix A.

Fig. 5 presents the syntax of the specification language, where $\kappa_1 * \kappa_2$ presents two non-overlapping heaps via separation conjunction $*$; $x.f \mapsto e$ means the field f of x points to e and $x:C$ means the run-time type of x stored in the heap is C. We use a simplified notation $x \mapsto C\langle \bar{f} : \bar{e} \rangle$ to denote a constructed heap object. $x \mapsto C\langle \bar{f} : \bar{e} \rangle$ is a point-to predicate where the object x has the exact type C and the fields $\bar{f}$ from C points to $\bar{e}$. We may shorten it to $x \mapsto C\langle \bar{f} \rangle$ for simplicity in some following sections. By default, we know which class a field f belongs to. Lastly, ϕ stands for pure arithmetic constraints. In contrast to the prior works

$$
\begin{array}{rcl}
p, q, f, m & ::= & (\kappa \wedge \phi) \mid p \vee p \mid \exists x.p\\
\kappa & ::= & emp \mid x.f \mapsto e \mid x : C \mid x \mapsto C\langle \bar{f} : \bar{e} \rangle \mid \kappa_1 * \kappa_2\\
\phi & ::= & true \mid false \mid x{=}e \mid x{<}e \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \Rightarrow \phi_2 \mid\\
 & & C_1{=}C_2 \mid C_1 \prec C_2 \mid ty(x){=}C \mid ty(x) \prec C \mid ty(x){\in}\{C_1, \ldots, C_n\}
\end{array}
$$

Fig. 5. An Assertion Logic for OOP

[35,21], we do not have the notation $x\not\mapsto$ called "negative heap", as we do not have explicit memory management, such as *free*() to de-allocate objects from heaps. In addition, we have a set of extra terms to constrain the types in our pure logic. The type of an object is immutable throughout its lifetime. We can use those terms to constrain the allocated type. For example, $ty(x){=}C$ means the run-time type of x is exactly C while $ty(x){\prec:}C$ ($ty(x){=}C \vee ty(x) \prec C$) can be used when x's type is either C or its subclasses.

4 Specification Inference

We semantically define IL triples [31] via program transitions. A configuration is a pair (S, σ) where S is a program and σ is a program state, i.e., the valuation of both memory stacks and heaps. A program transition is a binary relation $\rightsquigarrow$ on configurations. Relation $(S, \sigma) \rightsquigarrow (S', \sigma')$ holds if the execution of the statement in the configuration (S, σ) results in the new configuration (S', σ'). We define $\rightsquigarrow^*$, the reflexive-transitive closure of $\rightsquigarrow$, to capture finite executions. We assume all terminating executions end at a *skip* statement. We use $\sigma \in [\![p]\!]$ to denote that the program state σ satisfy the assertion p. Finally, $T_{sp} \vDash [p]\ S\ [\epsilon{:}q]$ denotes a valid IL triple, where T_{sp} is a context storing the specifications for the analyzed methods. Formally,

$$T_{sp} \vDash [p]\ S\ [\epsilon{:}q] \ \ \textit{iff}\ \ \forall \sigma.\, \sigma \in q,\ \ \exists \sigma'.\, \sigma' \in p\ \ s.t.\ \ (S, \sigma') \rightsquigarrow^* (\textit{skip}, \sigma)$$
$$\textit{with the specification context}\ T_{sp}\ \textit{and}\ \epsilon \in \{ok, er\}.$$

4.1 IL Triples For OO Statements and Type Constraint Propagation

Fig. 6 presents a set of valid IL triples for primitive OO program statements. As these triples hold without context T_{sp}, we omit it here. Rules **Skip**, **Read** and **Write** are standard. Rule **Assume** allows us to back-propagate the Boolean expression to the precondition as a path condition. There are three possibilities for the *instanceof* operation. Rule **InsNull** states that *null* is not an instance of any class. If x is allocated, it can either be or not be an instance of C, denoted

$\vDash [emp]\, skip\, [ok{:}\ emp]$ **Skip** $\quad \vDash [x.f{\mapsto}e_1 \wedge y{=}e_2]\ y{:=}x.f\ [ok{:}\ x.f{\mapsto}e_1 \wedge y{=}e_1]$ **Read**

$\vDash [x{=}null]\ y{:=}x.f\ [er{:}\ x{=}null]$ **NullRead** $\quad \vDash [x.f{\mapsto}e]\ x.f{:=}\ y\ [ok{:}\ x.f{\mapsto}y]$ **Write**

$\vDash [x{=}null]\ x.f{:=}y\ [er{:}\ x{=}null]$ **NullWrite** $\quad \vDash [emp{\wedge}b]\ assume(b)\ [ok{:}\ emp{\wedge}b]$ **Assume**

$\vDash [emp]\ error()\ [er{:}\ emp]$ **Error** $\quad \vDash [x{=}null]\, x\ instanceof\ C\ [ok{:}\ x{=}null{\wedge}\neg res]$ **InsNull**

$\vDash [x{\neq}null \wedge ty(x){\prec:}C]\ \ x\ instanceof\ C\ [ok{:}\ x{\neq}null \wedge ty(x){\prec:}C \wedge res]$ **InsT**

$\vDash [x{\neq}null \wedge ty(x){\not\prec:}C]\ \ x\ instanceof\ C\ [ok{:}\ x{\neq}null \wedge ty(x){\not\prec:}C \wedge \neg res]$ **InsF**

$\vDash [x{=}null]\ (C)\ x\ [ok{:}\ x{=}null \wedge res{=}x]$ **CastNull**

$\vDash [x{\neq}null \wedge ty(x){\prec:}C]\ (C)\ x\ [ok{:}\ x{\neq}null \wedge ty(x){\prec:}C \wedge res{=}x]$ **CastOk**

$\vDash [x{\neq}null \wedge ty(x){\not\prec:}C]\ (C)\ x\ [er{:}\ x{\neq}null \wedge ty(x){\not\prec:}C]$ **CastEr**

Fig. 6. Primitive IL Triples For OO Statements

```
1 public synchronized boolean equals (final Object other) {
2 [...ty(other) ≺: Object ∧ ty(other) ≺: AbsHis ∧ ty(other) ⊀: DblHis ]
3 if (other instanceof AbsHis) {
4 [ok: ...ty(other) ≺: Object ∧ ty(other) ≺: AbsHis ∧ ty(other) ⊀: DblHis ]
5    DblHis otherHis = (DblHis) other;
6 [er: ...ty(other) ≺: Object ∧ ty(other) ≺: AbsHis ∧ ty(other) ⊀: DblHis]
```

Fig. 7. Finding a CCE [8], via Type Constraint Propagation

by rules **InsT** and **InsF**. Similarly, there are three possibilities for casting, and one of them leads to CCEs. We use the default *res* in poststate q to denote the result being returned from an expression e in $[p]e[\epsilon{:}q]$.

Based on primitive IL triples, specification inference allows us to generate specifications for bigger code blocks [21,37], which consist of primitive statements. We show that such a mechanism can be applied to propagate type constraints according to program statements, which are critical for analysing OO programs. For example, statement *if* $(x$ *instanceof* $C)$ ... *else* ... results two possible specifications: $ty(x){\prec}{:}C$ for the if branch and $ty(x){\not\prec}{:}C$ for the else branch. Type constraints indicate the possible types for an object at run-time.

The example in Fig. 7 is taken from an open-source project HdrHistogram and fixed by the developer [8]. For simplicity, we only show the typing part of the inferred specification. The initial state before the *if* statement is $\phi_0 = (ty(other){\prec}{:}Object)$, obtained from the method signature. The (boxed) constraint $\phi_1 = (ty(other){\prec}{:}AbsHis)$ (according to the *if* condition) is back propagated to form the precondition for entering the *if* branch, i.e., $\phi_0 \wedge \phi_1$. For the casting operation at line 5, ToolX infers $\phi_2 = (ty(other) \not\prec{:} DblHis)$ (*highlighted*) as the missing formula which leads to an error postcondition. As the accumulated type constraint is satisfiable when reaching the *post*, i.e., $(\phi_0 \wedge \phi_1 \wedge \phi_2) \neq false$, it indicates that this error is on a feasible path. The states at line 2 and line 6 form an error specification for this method. In fact, *AbsHis* and *DblHis* are two unrelated classes, and this bug was caused by a typo from the programmer.

4.2 Inference Relations

We now discuss how to automatically achieve IL specification inference for OO programs. Given a statement S, we use the following relation $T_{sp} \vdash [p]?[M]\ S\ [\epsilon{:}\,q]$ to infer a missing formula M which is necessary to execute S and computes the corresponding postcondition $[\epsilon{:}\,q]$ with a given precondition p. T_{sp} is the specification table which initially contains the primitive rules in Fig. 6. For each analysed method, its inferred specifications are stored in T_{sp}, and used to further infer the specifications for the rest of the methods. Instead of using a standard $emp \wedge true$ symbolic heap [13,21] when analysing a new method, the inference is initialized with a precondition p that records the declared type for each input object. For example, given a method definition of class C: $t_0\ mn\ (args)\ \{S\}$, the

precondition for reasoning S is initialized as follows:

$$p = (\bigwedge_{(C'\ x)\in args} (x{=}null \vee ty(x) \prec: C')) \wedge ty(this) = C$$

The rest of the inference relations are presented in Fig. 8. The system performs forward symbolic executions. During the inference, the bi-abduction obligations in the form of $q * f \vdash_{bi} p * m$ are solved by the approaches in [13], where the missing resource m is inferred through frame calculation, and the *anti-frame* f carried is abduced. **ASSIGN-VAR** performs standard Floyd's forward assignment rule. **LOCAL** picks fresh variables to represent locally declared variables in specifications. The "default_value(v_t)" means the default value when a variable of type t is declared. **CHOICE** rule is design for non-deterministic choice + which paths could be split. **SEQ** performs the sequential composition. In **SEQ2**, $mod(S)$ returns the set of variables modified in the program S and $fv(f)$ is the set of free variables in formula f. The **UNROLLING** rule is designed Kleene star iterations S^* which allows it to unroll non-deterministically. In this work, we use upper-

ASSIGN−VAR
$$\frac{vars = (\bigwedge_{\forall y_i.y_i\in pvar(e)} y_i = e_i) \wedge x = x' \quad vars * f \vdash_{bi} p * m \quad q{=}\exists x'.m[x'/x] * p[x'/x] \wedge x{=}e[x'/x]}{T_{sp} \vdash [p]?[m]\ x{:=}e\ [ok{:}\ q]}$$

LOCAL
$$\frac{fresh(o) \quad \text{default\_value}(v_t) \quad T_{sp} \vdash [p \wedge o = v_t]?[m]\ S[o/x]\ [\epsilon{:}\ q]}{T_{sp} \vdash [p]?[m]\ t\ x; S\ [\epsilon{:}\ \exists o.\ q]}$$

VAL−VAR
$$\frac{q = p \wedge res = v}{T_{sp} \vdash [p]?[m]\ v\ [ok{:}\ q]}$$

CHOICE1
$$\frac{T_{sp} \vdash [p]?[m]\ S_1\ [\epsilon{:}\ q]}{T_{sp} \vdash [p]?[m]\ S_1 + S_2\ [\epsilon{:}\ q]}$$

CHOICE2
$$\frac{T_{sp} \vdash [p]?[m]\ S_2\ [\epsilon{:}\ q]}{T_{sp} \vdash [p]?[m]\ S_1 + S_2\ [\epsilon{:}\ q]}$$

SEQ1
$$\frac{T_{sp} \vdash [p]?[m]\ S_1\ [er : q]}{T_{sp} \vdash [p]?[m]\ S_1; S_2\ [er : q]}$$

SEQ2
$$\frac{T_{sp} \vdash [p_1]?[m_1]\ S_1\ [ok : q_1] \quad T_{sp} \vdash [p_2]?[m_2]\ S_2\ [\epsilon{:}:q_2] \quad (p_2 * m_2) * f \vdash_{bi} q_1 * m \quad mod(S_1) \cap fv(m) = mod(S_2) \cap fv(f) = \emptyset}{T_{sp} \vdash [p_1]?[m_1 * m]\ S_1; S_2\ [\epsilon{:}\ q_2 * f]}$$

CONSEQUENCE
$$\frac{p' \Rightarrow p \quad T_{sp} \vdash [p']?[m]\ S\ [\epsilon{:}\ q'] \quad q \Rightarrow q'}{T_{sp} \vdash [p]?[m]\ S\ [\epsilon{:}\ q]}$$

FRAME
$$\frac{T_{sp} \vdash [p]?[m]\ S\ [\epsilon{:}\ q] \quad mod(S) \cap fv(f) = \emptyset}{T_{sp} \vdash [p * f]?[m]\ S\ [\epsilon{:}\ q * f]}$$

UNROLLING
$$\frac{T_{sp} \vdash [p]?[m]\ skip + (S; S^*)\ [\epsilon{:}\ q]}{T_{sp} \vdash [p]?[m]\ S^*\ [\epsilon{:}\ q]}$$

ERR−CALL
$$\frac{m = (x = null)}{T_{sp} \vdash [p]?[m]\ x.mn\ [er : p]}$$

CALL-STATIC
$$\frac{ty\_constraints(x) \implies ty(x){=}C \quad ST(C.mn(\bar{w})) = ([p'] \_ [\epsilon{:}\ q']) \in T_{sp} \quad p'[x/this, \bar{y}/\bar{w}] * f \vdash_{bi} p * m \quad q = q'[x/this, \bar{y}/\bar{w}]}{T_{sp} \vdash [p]?[m]\ x.mn(\bar{y})\ [\epsilon{:}\ q * f]}$$

CALL-DYNAMIC
$$\frac{DY(C : mn(\bar{w})) \in T_{sp} \quad DY(C : mn(\bar{w})) \wedge ty\_constraints(x) = [p'] \_ [\epsilon{:}\ q'] \quad p'[x/this, \bar{y}/\bar{w}] * F \vdash_{bi} p * m \quad q = q'[x/this, \bar{y}/\bar{w}]}{T_{sp} \vdash [p]?[m]\ x.mn(\bar{y})\ [\epsilon{:}\ q * f]}$$

Fig. 8. Specification Inference Relations

bounded loop unrolling. The inference process terminates once it reaches an *er* postcondition.

Method Calls. There are two kinds of calls: CALL-STATIC and CALL-DYNAMIC are for static and dynamic calls, respectively. In our language, both the method calls are in the form of $x.mn(\bar{y})$ (we omit the arguments and use $x.mn$ for simplicity). We use $ty\_constraints(x)$ to denote the set of all type assertions of x in the pre-state formula. We say that this call can be statically determined if there is only one type possibility for x. For example, x is locally initialized by $new\ C(...)$, then $ty(x) = C$. In this case, we use the static specification for this call. Static specifications are directly inferred through the inference relations for each method by analysing its concrete implementation. We store the inferred specifications in T_{sp} and can be retrieved by $ST(C.mn)$. Note that we use CALL−STATIC to process the primitive statements shown in Fig. 6.

Dynamic Specifications. On the other hand, if the type of x is not statically determined, $x.mn$ is dynamically dispatched. We use $C.mn$ for the mn implementation in class C, and $C:mn$ to denote the set of mn implementations in C and its subclasses. Specifications for such $C:mn$ are dynamic specifications, denoted by $DY(C:mn)$. A natural way to derive dynamic specifications is to collect the static specifications of mn in all C', where $C' \prec: C$. Formally,

Definition 1. *Given class C and its subclasses, let $C:mn$ be the set of implementations of mn in these classes. The dynamic specification, denoted by $DY(C:mn)$, is defined as follows:*

$$DY(C:mn) = \bigwedge_{\forall C' \prec: C} ST(C'.mn).$$

The derived dynamic specifications will also be stored in T_{sp}. To find a correct dynamic specification for a dynamically dispatched call $x.mn$, we need to follow these steps: 1) Find the least positive type constraint of x (we call a type constraint $ty(x) \prec: C$, $ty(x) \not\prec: C$ as positive constraint and negative constraint, respectively). Let it be $ty(x) \prec: C_l$. By least positive type constraint, we mean that C_l is not the superclass of any other C in the other positive type constraints; 2) Find $DY(C_l:mn)$; 3) Trim $DY(C_l:mn)$ by removing specifications of infeasible types according to the negative type constraints. We show an example in Appendix C.1.

Note that constructors are special methods that only require static specifications. When analysing a constructor $C(...)$, the initial precondition p contains an allocated heap object (all uninitialized fields are null at the beginning) with the exact type $ty(...) = C$. Upon an *ok* termination, its reference is implicitly returned. We define the soundness of our inference mechanism in Theorem 1.

Theorem 1 (Soundness of the Inference Relations). *For all $T_{sp}, p, M, S, \epsilon, q$, if the inference relations conclude that $T_{sp} \vdash [p]?[M]\ S\ [\epsilon: q]$, then $T_{sp} \models [p * M]\, S\, [\epsilon: q]$ is a valid IL triple.*

Proof. *The proof consists of the soundness of both the inferred static and dynamic specifications. The details are provided in Appendix B.1.*

5 Bug Reporting

We aim to create a practical analyser with low false positives and high true positives. This section outlines our efforts to achieve this for OO programs.

5.1 Merging

Prior work [21] defines manifest bugs and latent bugs. In a nutshell, latent bugs are context-dependent, which will not always occur. In contrast, manifest bugs occur regardless of the calling context and should be reported to the user. In particular, to find manifest bugs, the previous tool classifies an *er* triple as manifest if its precondition is $emp \wedge true$ or *relaxed-manifest* if its precondition contains heap-allocated variables without any pure constraints. Otherwise, it is classified as a latent bug. However, this approach only reports a subset of manifest bugs as they examine specifications individually and hence may miss manifest bugs amongst multiple paths. We show such an example in Fig. 9, where class B extends class A, and two branches are rejoining at the *error*() statement. Hence, the error occurs regardless of the type of the input x.

```
1 void goo(A x) {
2   if (x instanceof B){skip;}
3   else {skip;}
4   error();}
```

Fig. 9. A Manifest Bug

We may infer two specifications for each branch separately. The error occurs in both the if branch and the else branch. However, using the previous approach, we will find that the inferred specifications contain path conditions $ty(x) \prec: B$ and $ty(x) \not\prec: B$, respectively. Therefore, we need to classify the triples in both branches as latent bugs and not report them to the user. To reduce such false negatives, we propose a merging mechanism which can join the *preconditions* of the specifications for the two branches so that this bug can be classified as a manifest bug.

On the other hand, the construction of dynamic specifications requires capturing specifications from multiple classes, which leads to path explosion for method calls. An under-approximating analyser will drop excessive specifications once the limit is reached. Although sacrificing precision, path dropping helps achieve scalability. Our merging mechanism can combine static specifications to form a more concise dynamic specification without losing path information. By doing this, we can slow down the path growth. Therefore, we enhance analysis precision via merging from two perspectives: 1) merging *preconditions* from error specifications to find more true bugs; and 2) merging static specifications to form dynamic specifications and slow down the path dropping.

Merging Mechanism We first defined *c-hierarchy* predicate in Definition 2 to model the class hierarchy in OO programs. Each *c-hierarchy* predicate has a tree-like structure where T is its root (superclass) with some subtrees (subclasses). A *c-hierarchy* predicate can model the full/partial class inheritance.

Definition 2 (*C-hierarchy* Predicate). *A* c-hierarchy *predicate is a disjunctive set of objects in the following form:*

$$D := \emptyset \mid T(\bar{f}, \bar{D})$$

A non-empty c-hierarchy *predicate pointed by* x *is defined as follows:*

$$x \mapsto T(\bar{f}, \bar{D}) \stackrel{def}{=} x \mapsto T\langle \bar{f} \rangle \ \vee \bigvee_{T_i(\bar{f}_i, \bar{D}_i) \in \bar{D}} x \mapsto T_i(\bar{f}{+\!+}\bar{f}_i, \bar{D}_i)$$

Recall that $x \mapsto T\langle \bar{f} \rangle$ indicates that x points to a heap object with exact type T. For $T(\bar{f}, \bar{D})$, T is the superclass name, $\bar{f}$ are the field mappings from T, and $\bar{D}$ is the predicates of some other classes directly extending T. The notation ++ is the appending operator. The subclasses (e.g., D_i) in a *c-hierarchy* predicate must always maintain the same state for field mappings inherited from the superclass (e.g., T). For example, $x \mapsto T_1(1, \{T_2(), T_3(2)\})$ means $x \mapsto T_1\langle 1 \rangle \vee x \mapsto T_2\langle 1 \rangle \vee x \mapsto T_3\langle 1, 2 \rangle$. A well-formed *c-hierarchy* predicate should respect the original class hierarchy from the program. Specifically, one *c-hierarchy* predicate must form a connected subgraph of the class hierarchy.

$$\frac{\begin{array}{c} S \prec_d T \\ var \mapsto S(\bar{f}_T{+\!+}\bar{f}_S, \bar{D}_S) * F * F' \quad \vee \quad var \mapsto T(\bar{f}_T, \bar{D}_T) * F * F_{@\bar{S}'} \end{array}}{var \mapsto T(\bar{f}_T, \bar{D}_T{+\!+}S(\bar{f}_S, \bar{D}_S)) * F * F_{@\bar{S}'} * F'_{@S}} \ \textbf{(Merging)}$$

This rule merges two formulae where *var* points to either a subclass or superclass *c-hierarchy* predicate, where $S \prec_d T$ means T is the direct superclass of S. The formula for the subclass S may contain an extra frame F' when *var* points to a subclass instance (e.g., the objects pointed by extension fields of subclasses). We tag this extra frame as $F'_{@S}$ to denote that F' is exclusively owned by the S *c-hierarchy* predicate after merging. Similarly, the formula for the superclass T may have already merged with some other direct subclasses $\bar{S}'$. Hence, it may contain some other tagged frames $F_{@\bar{S}'}$. These tagged frames will remain unchanged.

Note that the OO method's specifications will include *this* object, which denotes the current object. We merge two specifications from the superclass and the subclass using the above **Merging** rule for both *pre* and *post* by replacing *var* with *this*. This merging rule only merges formulae with the same F. In other words, we only merge the superclass and subclass specifications under the same path condition. We keep the specifications separate if the *pre* or *post* cannot be merged.

Merging makes the dynamic specification concise by simplifying a disjunctive form $P_1 \vee P_2$ to P_3 such that $P_3 = (P_1 \vee P_2)$ without loss of information. In the OO context, this happens quite often as a subclass usually behaves very similarly to its superclass. We illustrate the merging through the example in Fig. 10. Both *DblA* and *C* extend *A* where *DblA* overrides the original methods with a backup field to store the original value in field *val*. We infer static specifications for the three classes respectively: $[this \mapsto A\langle e \rangle]\_[ok\colon this \mapsto A\langle x \rangle]$ for *A*; $[this \mapsto DblA\langle e, b \rangle]\_[ok\colon this \mapsto DblA\langle x, e \rangle]$ for

```
1  class A {
2    int val;
3    void set(int x){
4      this.val = x;}}
5
6  class DblA extends A{
7    int bak;
8    void set(int x){
9      this.bak=this.val;
10     this.val = x; }}
11
12 class C extends A {}
```

Fig. 10. A Merging Example

DblA; and $[this \mapsto C\langle e\rangle]\_[ok\colon this \mapsto C\langle x\rangle]$ for C.
By using merging, the dynamic spec for $A : set$ can be obtained as:
$[this \mapsto A(e, \{DblA(b), C()\})]\_[ok\colon this \mapsto A(x, \{DblA(e), C()\})]$. Next, we define the generalised relaxed-manifest bug via merging.

Definition 3 (Relaxed-Manifest Bug). *Let E be a mapping from error statement s to the set of error specifications terminated at it. Then, s denotes a manifest bug if the following holds:*

- $E(s) \neq \emptyset$ *and* $\forall spec \in E(s).\ sat(post(spec))$
- $\forall spec \in E(s).\ pre(spec) \xrightarrow{merging\ steps} E_{pre}(s)$
- $\exists p \in E_{pre}(s).\ \ \kappa \wedge \phi_{ty} \vdash p$

Where κ is the heap formula representing the possible heap resources without pure constraints. ϕ_{ty} represents the initial type constraints (constructed from the initial method signature) we mentioned earlier.

We require the postconditions in specifications to be satisfiable and $E(s)$ is non-empty. $E_{pre}(s)$ is the set of formulae formed by merging the *preconditions* from all specifications in $E(s)$ through the following steps repeatedly until the preconditions cannot be merged.

- Step 1: Merge all *vars* in *pres* with the same path condition by **merging** rule.
- Step 2: Combine the merged formulae using the $\vee$ calculus.

These steps are trying to check if an error happens in several paths. "Context-independent" bugs in the OO program should occur regardless of the types of input parameters as the types of input objects are the additional dimension of the calling context. In the actual implementation, we sometimes relax this requirement. If there is no *instanceof* or casting throughout the method, we will report a bug that occurs when the types of inputs are the same as the declared ones since programmers may not consider subclasses in this case.

Note that the merging for the dynamic specification formation and bug reporting are different. The former is the merging of multiple *specifications* across multiple methods (from different classes) while the latter happens within one method and we only merge *preconditions*. Both of them may need to use *c-hierarchy* predicates to represent heap objects.

5.2 Reporting Class-Cast-Exceptions

A statement $(C)\ e$ could cause a CCE if $ty\_constraints(e) \Longrightarrow (ty(e) \prec: C)$. CCEs and NPEs share the following similarities: 1) The statement might not always trigger a runtime exception; 2) A guard can prevent the error (e.g., null checks for NPEs and *instanceof* checks for CCEs); 3) Without a guard, it's difficult to determine if an error should be reported, as the programmer may intentionally omit it based on their design, leading to potential false positives.

Since NPEs and CCEs exhibit similar characteristics, we can adopt the same methodology used for NPEs when addressing CCEs. However, our experimental findings indicate that this approach results in minimal detection of CCEs in real-world projects. There are two potential explanations for this. Firstly, programmers might not experience CCEs like NPEs; for instance, they may not pass a manifest-error object with incompatible types to methods. Secondly, CCEs could arise from external libraries with inaccessible source code or through code reuse. Programmers may lack awareness of the complete class hierarchy, leading them to overlook certain input object possibilities while coding. Even though only a certain kind of inputs can lead to CCEs, they could be in the interests of the programmers. According to a prior survey [30], 50% of the casting operations are unguarded by the *instanceof* checking which risks the programs. Is the casting operation safe when the programmers are aware of using *instanceof* checking? Our primary thought is that if CCEs still occur when programmers realise to do type filtering by using *instanceof*, it might be a mistake and we should alert the programmer about such a mistake. When we apply this strategy, we find some true CCEs in real-world projects, such as the examples in Fig. 1 and Fig. 7. We formally define the reporting criteria for CCEs in Definition 4.

Definition 4 (CCE Reporting Criteria). *An er triple is reportable if: It ends at a casting operation* $C\,(e)$ *and the postcondition is satisfiable; and*

- *It satisfies Definition 3; or*
- e *is an initialized object such that* $ty(e) = C'$ *and* $C' \not<: C$*; or*
- *An instanceof operator has been applied on* e *before the casting operation.*

6 Implementation and Evaluation

Implementation. We build ToolX inside Infer's framework (version id: 5050294) with an additional 10K lines of OCaml codes. We utilise Infer's bi-abductive entailment solver to compute missing formulae and frames. ToolX is an under-approximating analyser for finding bugs in Java programs. It performs compositional reasoning and generates IL triples for error reporting. In particular, ToolX includes a function $compute(p, T_{sp}, mn(\bar{C}\ \bar{o}))$ as the predicate transformer. Given a method mn, this function takes the initial precondition p mentioned in Sect. 4.2 and the specification table as inputs. It then applies the inference relations in Sect. 4 to infer the preconditions and the postcondition ϵ': Q' of mn. Given a Java program, ToolX first generates static specifications for methods and then, ToolX reports bugs on error triples if they satisfy the criteria in Sect. 5. The dynamic specifications are computed on-demand to save resources i.e., ToolX infers dynamic specifications for a method only when the method is dynamically dispatched and called somewhere. The inferred specifications are stored in T_{sp}.

To reduce the possible high cost from satisfiability checking when merging formulae for error reporting, we inspect errors which are likely to be manifest after merging, i.e., the error specifications occupy a large portion of paths when no path dropping. We use syntactic checks to filter pairs of triples that are more likely to be merged successfully. Using these heuristics, ToolX keeps more informative IL triples to assist with reportable bugs.

Evaluation. To conduct the experiments, we select a set of real-world programs as our benchmarks. In particular, the benchmarks are from a test case repository developed and maintained by Meta/Infer developers [18], Apache projects[1] and some popular code repositories which receive thousands of stars on Github. This Infer's repository contains challenging test cases and is accumulated in a real-world codebase. Some are for regression testing, and others for designing and testing new features of its tools, such as Pulse. The latter is beyond its capability, such as detecting CCEs. The experiments are designed to answer the following three research questions (RQ):

- RQ1: Is our approach capable of detecting CCEs in OO programs?
- RQ2: Are the detected CCEs containing false positives?
- RQ3: How does ToolX compare in performance with the state-of-the-art tool for detecting NPEs.

Table 1. CCEs Reported by ToolX and Error Prone. CCEs: number of CCEs reported. Fixed: the number of CCEs has been fixed according to the commits. Risky: the number of risky CCEs that have not been fixed in any commits. T: running time in seconds. The numbers in **red** indicate the false positives reported by ToolX.

#	Project		ToolX				Error Prone			
	Name	**KLoc**	**CCEs**	**Fxied**	**Risky**	**T**	**CCEs**	**Fxied**	**Risky**	**T**
1	Infer-c2dc303	11.4	2	0	2	11	0	0	0	2
2	pdfbox-a51dd40	12.1	4**+1**	2	2	42	0	0	0	28
3	ebean-b450227	20.7	3	0	3	40	3	0	3	42
4	HdrHistogram-9866a4c	27.2	1	1	0	27	1	1	0	5
5	jedis-febc027	33.9	1	1	0	20	0	0	0	12
6	spoon-9c1c3bf	46.5	6**+1**	2	4	43	0	0	0	33
7	classgraph-1310809180s	136.7	1	1	0	180	0	0	0	15
8	jfreechart-21922c1	292.6	1	0	1	32	0	0	0	30
9	Others	285.1	1**+2**	1	0	87	0	0	0	39
	Total	866.2	20**+4**	8	12	482	4	1	3	206

To answer RQ1, we summarize the experimental results on Table 1. Firstly, we compare the reported results with the Github commits. ToolX reports 24 CCEs in total and 8 (33.3%) are corrected by the developers. We examine the rest of the reports and find another 12 reports risky, especially when the code is used by someone unaware of the entire class hierarchy. Secondly, we compare ToolX with Error Prone (version 2.32.0), a popular static analyser developed by Google [2] for Java programs. Error Prone detects bugs through pattern recognition [3] and alerts users when the written code matches the pre-defined error patterns. Error Prone has reported four bugs which are the subset of ToolX's reports. One of the four is fixed by the developers while the other three match our risky reports. The results show that ToolX could effectively find more meaningful bugs in real-world programs.

To answer RQ2, as Table 1 shows, we conclude that there are 4 false positives. As the rules for reporting CCEs are designed to avoid false positives, the false

positive rate is fairly low (16%). We manually investigate the reports, such as by referring to the developer's comments or using semantic analysis. We find that although some bugs can be syntactically triggered, they may not be of users' interests. Hence, we mark them as false positives. We show a false positive in Fig. 11. According to our proposed reporting strategy, line 7 may contain a casting error. This is because ToolX finds out that there exist some types that are neither the subtype of *AnnotatedMethod* nor *AnnotatedField* for object *mutator*. Hence, casting at line 7 could be risky. It seems that the authors are aware of this issue and write a comment at line 6. The comment mentions that they should verify the correctness of this casting. However, the code has not been changed since the creation of this comment. Hence, *mutator* may not be an instance from the dangerous classes in an actual execution. It could be semantically safe.

Table 2. NPEs Reported by ToolX and Pulse. Op: the overlapping reports by both tools. $T_{TX,PL}$: running time in seconds. -FP: the number of false positives reduced by ToolX. +TP: the number of additional true bugs found by ToolX. +FP: the additional false positive reported by ToolX. -TP: the missed true bugs by ToolX. The commit ID of jackson-databind is 4a40123.

#	Project	KLoc	ToolX	T_{TX}	Pulse	T_{PL}	Op	-FP	+TP	+FP	-TP
1	Infer-c2dc303	11.4	96	11	89	10	89	0	7	0	0
2	pdfbox-606f916	21.6	48	44	50	41	44	3	4	0	3
3	spoon-5e77e89	33.5	9	101	11	96	6	5	2	1	0
4	ebean-b0ec23e	48.4	20	36	22	32	17	3	3	0	2
5	Botania-92f4863	77.4	21	47	18	39	18	0	3	0	0
6	ratis-8a50099	109.8	8	50	10	51	8	2	0	0	0
7	jackson-databind	210.3	6	47	12	18	6	6	0	0	0
8	picocli-a856a14	776.7	7	70	6	60	6	0	1	0	0
	Total	1279.6	215	406	218	347	194	19	20	1	5

To answer RQ3, we compare ToolX with Pulse (Infer version id: 5050294, July 2023). The results of our experiments are shown in Table 2. We analyse the bugs reported by both tools, categorizing them as true or false positives. Focusing on non-overlapping reports to highlight the differences between ToolX and Pulse, we find that ToolX eliminates an average of 16.9% of Pulse's false positives and identifies 10.1% new true positives. Together, these improvements

```
1 protected SettableBeanProperty constructSettableProperty(...){
2   ...
3   if (mutator instanceof AnnotatedMethod) {
4      prop = new MethodProperty((AnnotatedMethod) mutator);
5   } else {
6   // 08-Sep-2016, tatu: wonder if we should verify it is
        'AnnotatedField' to be safe?
7      prop = new FieldProperty((AnnotatedField) mutator);
8      // ToolX reports one error at line 7
```

Fig. 11. A (Simplified) False Positive Reported by ToolX [5]

lead to a 27.0% increase in precision. The missed true bugs and newly introduced false positives represent a small fraction of ToolX's reports and both tools exhibit similar running times. Overall, our findings demonstrate that our approach effectively enhances bug-finding precision.

7 Related Work and Conclusion

Incorrectness Logic. Applications of IL have been investigated in different domains, such as finding memory errors in large C projects [21], detecting data race/deadlock in concurrent programs [36], verifying quantum while-programs [16], detecting logical bugs in quantum programs [39], detecting forbidden graph structures and failing executions [34]. Similar to IL, other recent logics focusing on under-approximating reasoning include local completeness of abstract interpretation [12], outcome logic [40], and exact separation logic [29]. Unfortunately, none of them supports class inheritance and method overriding, except for [27]. [27] proposes a verification system for upholding Liskov substitution principle (LSP) in under-approximating reasoning. However, specifications must be manually provided in this system, and the lack of automation could limit its practicality in analysing large projects. Moreover, their work focuses on verification. It is hard for users to know if an error specification is risky or likely harmless. Our reporting criteria remedy this by only reporting dangerous error specifications automatically.

Formal Verification for OO programs. OO program verification via over-approximation has been extensively studied in various works: Verifying objects through dynamic frames to handle aliasing problem [19]; using supertype abstraction for concise and modular reasoning [28,24,22,23]; using separation logic and abstraction predicate for reasoning about abstract datatypes [32,38]; using class invariant to ensure the functional correctness of programs [17,9,26]. Later, two independent papers [14,33] propose the co-existence of static/dynamic specifications for OOP to uphold LSP while avoiding re-verification. Following the landscape of the proposals in [14,32,33,27], we propose our system for IL static/dynamic specification inference in OO programs.

Bugs in OO Programs. NPEs and CCEs are common bug types in OO programs. Error-prone is a pattern-based bug detector[2]. It supports CCE detection, but only finds CCEs in a specific way via pattern recognition [3]. In our work, we thoroughly study how to detect possible CCEs and our ToolX can effectively find more bugs. On the other hand, ToolX also outperforms another state-of-the-art Pulse in terms of finding NPEs as we model the OO features in our approach, such as class inheritance and method overriding. DOOP framework [11] performs pointer analysis for Java programs using Datalog, which potentially discovers CCEs when pointers are cast improperly. However, DOOP's analysis is not fully modular. It requires a *main* method as an entry point, and only pointers initialised can be checked. Such scenarios are the subsets of our CCE reporting criteria. DOOP could not find the errors like Fig. 1, Fig. 7.

Specification Inference via Bi-abduction. Bi-abduction [13] is a form of logical inference for separation logic that automates local reasoning. Bi-abduction generates *pre/post* based on *frame* and *anti-frame* formulae inference. Like the prior tool Pulse-X, we also make use of the bi-abduction technique in our specification inference process. Moreover, we incorporate type information analysis, which enables our tool to support class inheritance and method overriding. In addition, we propose the merging mechanism to support generalised error reporting, which improves the bug-finding precision.

Conclusion. Motivated by the question "How to generically and automatically infer IL specifications for object-oriented programs?", we demonstrate that carrying type information is crucial. Type constraints reveal runtime type possibilities, enabling static analysis of dynamic behaviours. Our system reasons about casting operations and infers static/dynamic specifications to effectively identify bugs in OO programs. Specifically, we formalise the *inference relations* to guarantee the validity of our inferred specifications. We also provide novel insights into bug reporting for OO programs, supporting both NPE and CCE detections through sound reasoning. Our approach establishes a formal foundation for IL-based bi-abductive inference in OO programs.

Acknowledgment. This work is supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001. We thank anonymous reviewers for their insightful comments.

References

1. The apache software foundation. https://github.com/apache. Accessed: 2024-1-13.
2. Error prone: a static analysis tool for java that catches common programming mistakes. https://github.com/google/error-prone. Accessed: 2024-05-16.
3. Error prone patterns: a static analysis tool for java that catches common programming mistakes. https://errorprone.info/bugpatterns. Accessed: 2024-05-16.
4. A false positive. https://github.com/apache/pdfbox/blob/trunk/debugger/src/main/java/org/apache/pdfbox/debugger/ui/ErrorDialog.java. Accessed: 2024-02-16.
5. jackson-databind. https://github.com/FasterXML/jackson-databind/blob/4a401237adfe3fd4e417504176171f76464aae96/src/main/java/com/fasterxml/jackson/databind/deser/BeanDeserializerFactory.java. Accessed: 2024-10-13.
6. An open source java tool for working with pdf documents: Pdfbox. https://github.com/apache/pdfbox/commit/eaf3b9862e80f1065f59acce150b38dd66a007c7. Accessed: 2024-02-16.
7. ToolX. https://zenodo.org/records/13934405.
8. Hdrhistogram - (commit id: 030aac1). https://github.com/HdrHistogram/HdrHistogram/commit/030aac1ea20b8c09e7c522a4594388534164d643, 12 2023. Accessed: 2023-12-20.
9. 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.

10. 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.
11. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009.
12. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *J. ACM*, 70(2), mar 2023.
13. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.
14. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 87–99, New York, NY, USA, 2008. Association for Computing Machinery.
15. Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie Van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 134–145. IEEE, 2015.
16. Yuan Feng and Sanjiang Li. Abstract interpretation, hoare logic, and incorrectness logic for quantum programs. *Information and Computation*, 294:105077, 2023.
17. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
18. Infer. Infer's test repository. `https://github.com/facebook/infer/blob/main/infer/tests/codetoanalyze/java/pulse`. Accessed: 2023-8-20.
19. 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.
20. Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 312–315, 2014.
21. 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.
22. Gary T Leavens and David A Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. *Department of Computer Science, Iowa State University, Ames, Iowa*, 50011:06–36, 2006.
23. 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.
24. Gary T Leavens and William E Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.
25. Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Npex: Repairing java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1532–1544, 2022.

26. K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004.
27. Wenhua Li, Quang Loc Le, Yahui Song, and Wei-Ngan Chin. Incorrectness proofs for object-oriented programs via subclass reflection. In Chung-Kil Hur, editor, *Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings*, volume 14405 of *Lecture Notes in Computer Science*, pages 269–289. Springer, 2023.
28. 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.
29. Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:27, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
30. Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. Casting about in the dark: An empirical study of cast operations in java programs. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019.
31. Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.
32. 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.
33. Matthew J Parkinson and Gavin M Bierman. Separation logic, abstraction and inheritance. *ACM SIGPLAN Notices*, 43(1):75–86, 2008.
34. Christopher M Poskitt. Incorrectness logic for graph programs. In *International Conference on Graph Transformation*, pages 81–101. Springer, 2021.
35. 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 *International Conference on Computer Aided Verification*, pages 225–252. Springer, 2020.
36. Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. Concurrent incorrectness separation logic. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.
37. Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. Provenfix: Temporal property-guided program repair. *Proc. ACM Softw. Eng.*, 1(FSE):226–248, 2024.
38. Stephan van Staden, Cristiano Calcagno, and Bertrand Meyer. Verifying executable object-oriented specifications with separation logic. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*, pages 151–174. Springer, 2010.
39. Peng Yan, Hanru Jiang, and Nengkun Yu. On incorrectness logic for quantum programs. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–28, 2022.
40. Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

A Semantics

Val defines values of variables including integers, booleans, locations *Loc*, and *null*. A program state $\sigma \in PState$ is a tuple, including a stack $s \in Stack$, that maps variables to values, *Val*, and a heap $h \in Heap$, 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.

$$\sigma \in PState \stackrel{\text{def}}{=} Stack \times Heap \qquad s \in Stack \stackrel{\text{def}}{=} Var \rightarrow Val \qquad v \in Val$$

$$h \in Heap \stackrel{\text{def}}{=} (Loc \rightharpoonup Classes) \times (Loc \times Field \rightharpoonup Val) \qquad l \in Loc \subseteq Val$$

$$[\![.]\!] \stackrel{\text{def}}{=} Statement \times Exit \times \mathscr{P}(PState \times PState) \qquad \epsilon \in Exit \stackrel{\text{def}}{=} \{ok, er\}$$

$[\![skip]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s,h))\} \qquad [\![x{:=}e]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[x{\mapsto}s(e)],h))\}$

$[\![x{:=}y.f]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[x{\mapsto}v],h))|\ h.2(s(y),f)=v\}$

$[\![x{:=}y.f]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ s(y)=null\}$

$[\![x.f{:=}y]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s,h'))|\ h'=(h.1,h.2[(s(x),f){\mapsto}s(y)])\}$

$[\![x.f{:=}y]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ s(x)=null\} \qquad [\![error()]\!]er \stackrel{\text{def}}{=} \{((s,h),(s,h))\}$

$[\![assume(B)]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s,h))|\ s(B)=True\}$

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

$B[\![x\ instanceof\ C]\!](s,h) \stackrel{\text{def}}{=} True\ \textit{iff}\ (h.1(s(x)) \prec C)$

$[\![y{:=}(C)\ x]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s[y\mapsto s(x)],h))|\ (h.1(s(x))=C_1 \wedge C_1{\prec}C) \vee s(x)=null\}$

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

$$[\![S_1;S_2]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|\epsilon=er \wedge ((s,h),(s',h')) \in [\![S_1]\!]er \\ \vee \exists(s'',h'').\ ((s,h),(s'',h'')) \in [\![S_1]\!]ok \wedge ((s'',h''),(s',h')) \in [\![S_2]\!]\epsilon\}$$

$[\![S_1+S_2]\!]\epsilon \stackrel{\text{def}}{=} \{[\![S_1]\!]\epsilon \cup [\![S_2]\!]\epsilon\}$

$[\![S^*]\!]\epsilon \stackrel{\text{def}}{=} \{\bigcup_{i\in\mathbb{N}} [\![S^i]\!]\epsilon\} \quad S^0=skip \ \ and \ \ S^{i+1}=S;S^i$

$[\![t\ x;S]\!]\epsilon \stackrel{\text{def}}{=} \{((s[x{\mapsto}v],h),(s'[x{\mapsto}v],h'))|((s,h),(s',h')) \in [\![S]\!]\epsilon\}$

$[\![return\ y]\!]ok \stackrel{\text{def}}{=} \{((s,h),(s',h))|\exists s''.\ s'=s''[ret{\mapsto}s(y)]\}$

$$[\![x.mn(\bar{z})]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|(\exists s_1.\ ((s_o[\bar{w}{\mapsto}s(\bar{z}),this\mapsto s(x)],h),(s_1,h')) \in [\![S]\!]ok \\ \wedge ((s_1,h'),(s'=s[ret{\mapsto}s_1(y)],h')) \in [\![return\ y]\!]ok) \\ \vee (\exists S', s_1.\epsilon=er \wedge ((s_o[\bar{w}{\mapsto}s(\bar{z}),this\mapsto s(x)],h),(s_1,h')) \in [\![S']\!]er \wedge s'=s) \\ \vee ((s',h')=(s,h) \wedge \epsilon=er \wedge s(x)=null)\}$$

provided that $h.1(s(x))=C$, $body(C.mn(\bar{w}))=\{S;return\ y\}$;
s_o is new method stack;
S' is a sub-sequence of statements (from beginning) of S

$$[\![new\ C(\bar{y})]\!]\epsilon \stackrel{\text{def}}{=} \{((s,h),(s',h'))|\ \exists l.\ loc(l) \notin dom(h.1) \\ \wedge l.1(loc(l))=C \wedge l.2[\overline{(loc(l),f)}{\mapsto}null] \wedge \\ ((\exists s_1.\ ((s_o[\bar{w}{\mapsto}s(\bar{y})],h \uplus l),(s_1,h')) \in [\![S]\!]ok \\ \wedge ((s_1,h'),(s'=s[ret{\mapsto}loc(l)],h')) \in [\![return\ loc(l)]\!]ok) \\ \vee (\exists S', s_1.\epsilon=er \wedge ((s_o[\bar{w}{\mapsto}s(\bar{y})],h \uplus l),(s_1,h')) \in [\![S']\!]er) \wedge s'=s))\}$$

provided that $body(C(\bar{w}))=\{S\}$, S' is a sub-sequence of S;
$loc(l)$ returns the location of l; s_o is new method stack

Fig. 12. Semantics of a Core OO Language.

Semantics of our core programs are defined in Fig. 12. Each method call will create a new stack s_0 (called method scope) [10] 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 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*.

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. 13.

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

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

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

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

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

Fig. 13. Semantics of Assertion Language.

B Extra notes

B.1 Soundness

Static Specification Inference Soundness The primitive rules in Fig. 6 are IL triples. We make some minor changes to make sure that it only contains the minimal resources for the instructions to be processed. The triples are proven to be sound in a prior work [27]. As we use the same semantics for the OO core language as the prior work, the IL triples remain valid in our setting. The inference relations via bi-abduction in Fig. 8 are designed to fit the semantics of the other commands in the core language. They ensure the soundness of specification inference by construction. The inference relations are the key system to perform the forwards-running symbolic execution.

Dynamic Specification Inference Soundness The following principle guarantees the soundness of dynamic specifications for $C : mn$ defined in Definition **??**.

Theorem 2 (Subsumption Principle). *A dynamic specification* $[P]\ C : mn\ [\epsilon{:}Q]$ *is valid if for all possible type* T *where* $T \prec: C$*, we have* $[P_T]\,T.mn[\epsilon{:}Q_T]$ *and* $[P]\_[\epsilon{:}Q]$ *satisfies the following subsumption,*

$$[P_T]\_[\epsilon{:}Q_T] <:_U [P]\_[\epsilon{:}Q \wedge type(this){=}T]$$

where $[P_T]\_[\epsilon{:}Q_T]$ *is the static specification for* $T.mn$ *and spec subsumption* $<:_U$ *is defined as,*

$$\frac{Q_2 \models Q_1 * F \qquad P_1 * F \models P_2}{[P_1]\_[\epsilon{:}Q_1] <:_U [P_2]\_[\epsilon{:}Q_2]}$$

Proof. $<:_U$ combines the consequence and the frame rules in IL. For any code satisfying the specification on the left-hand side $<:_U$, it satisfies the specification on the right-hand side. Suppose we have $[P_1]\_[\epsilon{:}Q_1] <:_U [P_2]\_[\epsilon{:}Q_2]$

$$\frac{\dfrac{[P_1]\_[\epsilon{:}Q_1]}{P_1 * F \models P_2 \quad [P_1 * F]\_[\epsilon{:}Q_1 * F] \quad Q_2 \models Q_1 * F}(Frame)}{[P_2]\_[\epsilon{:}Q_2]}(Consequence)$$

Proposition 1. *The dynamic specifications formed by conjuncting the static specifications follow the subsumption principle.*

Proof. According the derivation, it is a fact that $DY(T : mn)$ is a conjunctive set of $AT(C_i.mn)$ for all subclasses C_i. Hence, we have

$$ST(C.mn) \Leftrightarrow DY(A : mn) \wedge ty(this) = C \quad \text{where } C \prec: A$$

It is just a specifial case of Theorem 2 as the following,

$$[P_T]\_[\epsilon{:}Q_T] = [P]\_[\epsilon{:}Q \wedge type(this){=}T]$$

B.2 Merging Specifications

To reduce the number of paths of a dynamic specification, we propose a mechanism to merge compatible paths.

The derivation of dynamic specifications can be expressed using the **Disjunction** rule in IL, namely:

$$\frac{[p_1]S[\epsilon{:}q_1] \qquad [p_2]S[\epsilon{:}q_2] \qquad p_1 \wedge p_2 = \mathit{false}}{[p_1 \vee p_2]S[\epsilon{:}q_1 \vee q_2]}\ (\textbf{Disjunction})$$

Lemma 1. *Specification merging via* **Disjunction** *rule to form dynamic specifications is precise, i.e., we can derive the consequent from the antecedent and vice versa.*

Proof. The consequent can be derived from the antecedent using the standard disjunction rule in incorrectness logic [31].

As $[p_1]S[\epsilon{:}q_1]$ and $[p_2]S[\epsilon{:}q_2]$ are static specifications from different classes, we can view it as the following form.

$$[\mathit{this} \mapsto t_A * p_1]S[\epsilon{:}\mathit{this} \mapsto t_A * q_1]$$

$$[\mathit{this} \mapsto t_B * p_2]S[\epsilon{:}\mathit{this} \mapsto t_B * q_2]$$

As the type of *this* will never be changed throughout S, we can apply a type constraint to get the specification back, for example,

$$\frac{\dfrac{[\mathit{this} \mapsto t_A * p_1 \vee \mathit{this} \mapsto t_B * p_2]S[\epsilon{:}\mathit{this} \mapsto t_A * q_1 \vee \mathit{this} \mapsto t_B * q_2]}{[(\mathit{this} \mapsto t_A * p_1 \vee \mathit{this} \mapsto t_B * p_2) \wedge ty(\mathit{this}) = A]S}}{[\epsilon{:}(\mathit{this} \mapsto t_A * q_1 \vee \mathit{this} \mapsto t_B * q_2) \wedge ty(\mathit{this}) = A]}$$
$$[\mathit{this} \mapsto t_A * p_1]S[\epsilon{:}\mathit{this} \mapsto t_A * q_1]$$

We can apply the similar method to get $[\mathit{this} \mapsto t_B * p_2]S[\epsilon{:}\mathit{this} \mapsto t_B * q_2]$ and complete the proof.

Such a disjunctive form in the merged specification does not provide any real advantages over the separate specifications. Our solution is therefore to *only* do merging when the disjunctive form $P_1 \vee P_2$ can be simplified to a conjunctive form $(P_1 \vee P_2) \equiv P_3$ without loss of information. In the OO context, this happens quite often via method inheritance and overriding. *c-hierarchy* predicate helps to merge formulae. We design the following operations for *c-hierarchy* predicate.

$$\frac{S \prec_d T}{\mathit{var} \mapsto T(\bar{f}_T, \bar{D}_T) \vee \mathit{var} \mapsto S(\{\bar{f}_T, \bar{f}_S\}, \bar{D}_S)}{\mathit{var} \mapsto T(\bar{f}_T, \{\bar{D}_T, S(\bar{f}_S, \bar{D}_S)\})}\ (\textbf{Join})$$

where $S \prec_d T$ means T is the direct superclass of S. The **Join** operation connects mutiple types togather. Based on the **Join** operation, we can merge two formulae where *var* appears to be different in each formula. The formula for the subclass S may contain an extra frame F_2 (not reachable in T's formula). Other than the

Join, we have the extra following rules. On the other hand, we may only need a subset of a *c-hierarchy* predicate. To obtain a subset, we allow two operations over the *c-hierarchy* predicate.

$$\frac{var \mapsto T(\bar{f}_T, \bar{D}_T) \quad \exists\ S(\bar{f}_s, \bar{D}_s).\ S(\bar{f}_s, \bar{D}_s) \in T(\bar{f}_T, \bar{D}_T) \quad ty(var) \prec: S \quad \{C\} \to S}{var \mapsto S(((\bar{f}_{C_i})_{\forall C_i \in \{C\}}, \bar{f}_s), \bar{D}_s))}\ (\textbf{Extract})$$

The **Extract** operation can extract an inner dynamic predication from the original predicate. Suppose an object var is pointing to a *c-hierarchy* predicate $T(\bar{f}_T, \bar{D}_T)$. When the type constraint is applied to var, all type instances which do not satisfy the type constraint $ty(var) \prec: S$ will be removed from the *c-hierarchy* predicate. $\{C\} \to S$ means the set of all the superclasses of S in $T(\bar{f}_T, \bar{D}_T)$. The field information from all superclasses of S will be inherited by the remaining *c-hierarchy* predicate after removing.

$$\frac{var \mapsto T(\bar{f}_T, \bar{D}_T) \quad \exists\ S(\bar{f}_s, \bar{D}_s).\ S(\bar{f}_s, \bar{D}_s) \in T(\bar{f}_T, \bar{D}_T) \quad ty(var) \not\prec: S}{var \mapsto T(\bar{f}_T, \bar{D}_T) \backslash S(\bar{f}_s, \bar{D}_s)}\ (\textbf{Cut})$$

The **Cut** operation is pruning a *c-hierarchy* predicate in the opposite way. Specifically, it can be used to remove one *c-hierarchy* predicate from the original set when such predicate is not the instance of S. The above two operations are useful when there are type constraints imposed by $instanceof$, $\neg instanceof$ and casting. We show an example in Sect. C.1.

In addition, we also need to define where there no single *c-hierarchy* predicate in $T(\bar{f}_T, \bar{D}_T)$ satisfying the type constraint.

$$\frac{var \mapsto T(\bar{f}_T, \bar{D}_T) \quad \nexists\ S(\bar{f}_s, \bar{D}_s).\ S(\bar{f}_s, \bar{D}_s) \in T(\bar{f}_T, \bar{D}_T) \quad ty(var) \prec: S}{false}\ (\textbf{ExtractFalse})$$

Similarly, all the possible types are subtypes of S but the constraint requires $ty(var) \not\prec: S$. We also define it to be false in this case.

$$\frac{var \mapsto T(\bar{f}_T, \bar{D}_T) \quad T \prec: S \quad ty(var) \not\prec: S}{false}\ (\textbf{CutFalse})$$

Recall that a well-formed c-hierarchy predicate should respect the original class hierarchy from the program.

Proposition 2 (*c-hierarchy* predicate Well-formedness). *The* c-hierarchy *predicates obtained by* **Join**, **Extract** *and* **Cut** *are well-formed, i.e., the ordering of classes in* c-hierarchy *predicate respects to the class hierarchy.*

Proof. The *c-hierarchy* predicate is a disjunctive set for some objects from different classes. Every sub-predicate (e.g., D_i) is just a subset of the original *c-hierarchy* predicate. The three operations **Join**, **Extract** and **Cut** for *c-hierarchy* predicate directly follow the definition. The **Join** operation join two sets and the

constraint $S \prec_d T$ ensures the join predicate is still well-formed. This is because we only connect two predicates when one is directly extended by the other. The ordering of classes will be the same as the hierarchy obtained from the code. The operation **Extract** and **Cut** find subsets of a *c-hierarchy* predicate. As the **Join** operation will construct well-formed *c-hierarchy* predicate. **Extract** and **Cut** do not alert the ordering of the classes. The subset is still well-formed.

C Examples

C.1 Usage of Dynamic Specifications

Dynamic specifications may need to be pre-processed before applying to dynamic dispatching calls. This is because the possible types of program variables may be constrained by program statements, such as the *instanceof* operation. Consider the same example in Fig. 10, suppose we have the following method:

$$void\ foo(A\ a,\ int\ x)\{if\ (!(a\ instanceof\ DblA))\{a.set(x)\}\}$$

Because of the conditional, there are only two possible types, A and C for the dynamic dispatching call $a.set(x)$ while a remains unchanged if its type is B.

The specification of $a.set(x)$ can be extracted from $D(A : set)$ by applying $ty(a) \not\prec: DblA$ on both of the *pre* and *post* of $D(A : set)$. As the merging is precise according to Lemma 1, ruling out some types from a dynamic specification does not affect its soundness. We could get a specification for $a.set(x)$ through the following step.

$$\frac{\begin{array}{c}[this \mapsto A(e, \{DblA(b), C()\})]A : set[ok{:}\ this \mapsto A(x, \{DblA(e), C()\})] \\ \hline [this \mapsto A(e, \{DblA(b), C()\}) \wedge ty(this) \not\prec: DblA]A : set \\ \hline [ok{:}\ this \mapsto A(x, \{DblA(e), C()\}) \wedge ty(this)\ \not\prec{:} DblA] \\ \hline [this \mapsto A(e, C())]\ A : set\ [ok{:}\ this \mapsto A(x, C())]\end{array}}{}\ (\textbf{Frame})$$

These steps follow the **Frame** rule in IL. Type constraints are important as they allow us to identify the correct types of variables when we analyse the programs.

C.2 Specification Inference

Fig. 14 shows the set method in class A and its subclass B. The *highlight* formulae, in the greyed box background, are the missing resources inferred via our inference relations. The missing parts are propagated back to the precondition. In addition, the pure constraints generated from *assume* statement are boxed and sent back as part of the path condition. In Fig. 14, we show how to generate one static specification (st 1) for $A.set$. The second static specification, which works for another path, can be generated through the same procedures. As the object c is not accessed throughout the method, the type information is

```
1  class C {}
2  class A {
3      C f;
4      bool tag;
5      void set(C c){
6  [this ↦ A⟨ f : e2 , tag : e1 ⟩ ∧ e1 = true ∧ ty(c) ≺: C]
7  assume(this.tag);
8  [ok: this ↦ A⟨ f : e2 , tag : e1⟩ ∧ e1 = true ∧ ty(c) ≺: C]
9  this.f = c;
10 [ok: this ↦ A⟨f : c, tag : e1⟩ ∧ e1 = true ∧ ty(c) ≺: C]
11     }
12 }
13     st 1:[this ↦ A⟨f : e2, tag : e1⟩ ∧ e1 = true]_
14 [ok: this ↦ A⟨f : c, tag : e1⟩ ∧ e1 = true]
15     st 2:[this ↦ A⟨tag : e⟩ ∧ e = false]_
16           [ok: this ↦ A⟨tag : e⟩ ∧ e = false]
17     dy 1:[this ↦ A({f : e2, tag : e1}, B()) ∧ e1 = true]_
18           [ok: this ↦ A({f : c, tag : e1}, B()) ∧ e1 = true]
19     dy 2:[this ↦ A⟨tag : e⟩ ∧ e = false]_
20           [ok: this ↦ A⟨tag : e⟩ ∧ e = false]
21     dy 3:[this ↦ B⟨f : e2, tag : e1⟩ ∧ e1 = false]_
22 [ok: this ↦ B⟨f : null, tag : e1⟩ ∧ e1 = false]
23 class B extends A {
24     @override
25     void set(C c){
26  assume(this.tag);this.f = c;
27  + //choice
28  assume(this.tag);this.f = null;
29     }
30 }
31     st/dy 1:[this ↦ B⟨f : e2, tag : e1⟩ ∧ e1 = true]_
32     [ok: this ↦ B⟨f : c, tag : e1⟩ ∧ e1 = true]
33     st/dy 2:[this ↦ B⟨f : e2, tag : e1⟩ ∧ e1 = false]_
34     [ok: this ↦ B⟨f : null, tag : e1⟩ ∧ e1 = false]
```

Fig. 14. The set Method for Class A and B

erased from specifications (namely, ~~$ty(c) \prec: C$~~). The inference of static specifications for $B.set$ is similar. Hence, we omit the details.

Note that static specification one (st 1) of both $A.set$ and $B.set$ have the same path condition. Hence, we can merge them to form the dynamic specification one (dy 1).

$$\frac{\begin{array}{c} this \mapsto A\langle f : e_2, tag : e_1\rangle \wedge e_1 = true \\ this \mapsto B\langle f : e_2, tag : e_1\rangle \wedge e_1 = true \end{array}}{this \mapsto A(\{f : e_2, tag : e_1\}, B()) \wedge e_1 = true}(pre)$$

$$\frac{\begin{array}{c} this \mapsto A\langle f : c, tag : e_1\rangle \wedge e_1 = true \\ this \mapsto B\langle f : c, tag : e_1\rangle \wedge e_1 = true \end{array}}{this \mapsto A(\{f : c, tag : e_1\}, B()) \wedge e_1 = true}(post)$$

As the remaining two specifications could not be merged, they are kept as the original in (dy 2) and (dy 3). The dynamic and static specifications are the same in B as there is no subclass of B.

Let us consider the two methods in Fig. 15. Similarly, as c is not accessed in the method body, we remove the type information of c from specifications. We generate two possible paths here. For the first one, we add $ty(a) \prec: B$ to the spec after *assume* (a *instance* B). As $ty(a) \prec: B \Rightarrow ty(a) \prec: A$, we omit $ty(a) \prec: A$ in the following for simplicity. The inference rules compute a missing resource M for statement $a.tag = false$ at line 7 where M is $a.tag \mapsto e_1]$. Together with the type constraint $ty(a) \prec: B$, we can use a dynamic predicate to represent this as $a \mapsto B(\{tag : e\}, \emptyset)$. $a.set(c)$ at line 15 takes (st/dy 2) of $B.set$ to perform the method call according to the type constraint.

Similarly, when we perform the method call for the second path where $ty(a) \not\prec: B \wedge ty(a) \prec: A$. We extract the following specification from (dy 1).

$$\frac{\begin{array}{c} [this \mapsto A(\{f : e_2, tag : e_1\}, B()) \wedge e_1 = true]A : set \\ [ok{:}\ this \mapsto A(\{f : c, tag : e_1\}, B()) \wedge e_1 = true] \end{array}}{\begin{array}{c} [this \mapsto A(\{f : e_2, tag : e_1\}, B()) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B]A : set \\ [ok{:}\ this \mapsto A(\{f : c, tag : e_1\}, B()) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B] \end{array}}$$

$$\frac{\begin{array}{c} [this \mapsto A(\{f : e_2, tag : e_1\}, B()) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B]A : set \\ [ok{:}\ this \mapsto A(\{f : c, tag : e_1\}, B())) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B] \end{array}}{\begin{array}{c} [this \mapsto A(\{f : e_2, tag : e_1\}, \emptyset) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B]A : set \\ [ok{:}\ this \mapsto A(\{f : c, tag : e_1\}, \emptyset)) \\ \wedge e_1 = true \wedge ty(this) \prec: A \wedge ty(this) \not\prec: B] \end{array}}$$

We omit the error specification where $a = null$, which is caused by an NPE at line 12. This is because the bug we will discuss in method *error* is a relaxed manifest bug which assumes the input arguments are initialized. For method *error*, we generate two paths as the method call at line 26 produces two possibilities according to the specification in *apply*. The field f of a is null after the method calls for both paths. Hence, we have two error specifications as the call of *toString* on line 29 causes NPEs when the calling object is null. To report

```
...
void apply(A a, C c){
    [pre1 : a ↦ B({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ≺: B ]
    [pre2 : a ↦ A({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ⊀: B ]
    {assume (a instanceof B);
    [ok1: a ↦ B({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ≺: B]
     a.tag = false;
    [ok1: a ↦ B({ f : e2 ,tag : e},∅) ∧ e = false ∧ ty(a) ≺: B]
+ //choice
     assume (not (a instanceof B));
    [ok2: a ↦ A({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ⊀: B]
     a.tag = true;
    [ok2: a ↦ A({ f : e2 ,tag : e},∅) ∧ e = true ∧ ty(a) ≺: A
    ∧ty(a) ⊀: B]
     }
    a.set(c);
    [ok1: a ↦ B⟨f : null, tag : e⟩ ∧ e = false ∧ ty(a) ≺: B]
    [ok2: a ↦ A({f : c, tag : e},∅) ∧ e = true
∧ty(a) ≺: A ∧ ty(a) ⊀: B]
}

void error(A a){
    [pre1 : a ↦ B({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ≺: B ]
    [pre2 : a ↦ A({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ ty(a) ⊀: B ]
    C c = null;
    [ok1: ∃o.a ↦ B({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ o = null ∧ ty(a) ≺: B ]
    [ok2: ∃o.a ↦ A({ f : e2 , tag : e1 },∅) ∧ ty(a) ≺: A ∧ o = null ∧ ty(a) ⊀: B ]
    apply(a,c);
    [ok1: ∃o.a ↦ B⟨f : null, tag : e⟩ ∧ e = false ∧ ty(a) ≺: B ∧ o = null]
    [ok2: ∃o.a ↦ A({f : c, tag : e},∅)
∧e = true ∧ ty(a) ≺: A ∧ ty(a) ⊀: B ∧ o = null]
    a.f.toString();
    [er1: ∃o.a ↦ B⟨f : null, tag : e⟩ ∧ e = false ∧ ty(a) ≺: B ∧ o = null]
    [er2: ∃o.a ↦ A({f : o, tag : e},∅)
∧e = true ∧ ty(a) ≺: A ∧ ty(a) ⊀: B ∧ o = null]
}

...
```

Fig. 15. An Illustrative Example

this error, our system applies the *merges* mechanism for their preconditions and classifies the merged specification as a relaxed manifest bug as follows.

$$\frac{\begin{array}{l} a \mapsto B(\{f : e_2, tag : e_1\}, \emptyset) \wedge ty(a) \prec: A \wedge ty(a) \prec: B \quad \vee \\ a \mapsto A(\{f : e_2, tag : e_1\}, \emptyset) \wedge ty(a) \prec: A \wedge ty(a) \not\prec: B \end{array}}{\dfrac{\begin{array}{l} a \mapsto B(\{f : e_2, tag : e_1\}, \emptyset) \quad \vee \\ a \mapsto A(\{f : e_2, tag : e_1\}, \emptyset) \end{array}}{a \mapsto A(\{f : e_2, tag : e_1\}, B())}}$$

In the first step, we remove the redundant pure information from specifications. For example, as $a \mapsto B(\{f : e_2, tag : e_1\}, \emptyset) \Rightarrow ty(a) \prec: A \wedge ty(a) \prec: B$, we just keep $a \mapsto B(\{f : e_2, tag : e_1\}, \emptyset)$. The merging result shows that $\kappa \wedge \phi_{ty} \vdash a \mapsto A(\{f : e_2, tag : e_1\}, B())$. a is an allocated object without any pure conditions. ϕ_{ty} is the type information constructed from the method signature $ty(a) \prec: A$. It implies all types in the dynamic predicate $A(\{f : e_2, tag : e_1\}, B())$. Therefore, our system reports this error.