Separation, Abstraction, Multiple Inheritance and View Shifting

Stephan van Staden[1] and Cristiano Calcagno**[2]

[1] ETH Zurich, Switzerland
Stephan.vanStaden@inf.ethz.ch
[2] Imperial College, London
ccris@doc.ic.ac.uk

Abstract. Inheritance is a central mechanism in object-oriented programming. Many popular object-oriented languages support multiple inheritance or limited versions thereof. This work extends a powerful modular proof system for single inheritance, which uses separation logic and abstract predicate families, to multiple inheritance. The extended system allows view shifting in the logic: the ability to view an object under different abstractions and to shift between such views. Several examples illustrate the system's use and utility.

Key words: Object-orientation, Separation logic, Abstraction, Abstract predicate families, Multiple inheritance, View shifting

1 Introduction

Inheritance is a central concern in object-oriented program verification. Many languages offer multiple inheritance (e.g. C++ and Eiffel [1]) or limited versions thereof (e.g. Java interfaces). While multiple inheritance is more expressive, it is also harder to reason about.

Few logics exist for reasoning about multiple inheritance, while many program logics accommodate single inheritance. Parkinson and Bierman [3] recently presented a sound proof system for single inheritance with several useful properties: 1) it offers local reasoning despite aliasing, 2) the logic supports abstraction and information hiding well with abstract predicate families (abbreviated apfs), 3) the system is modular and every method body is verified only once, and 4) a wide range of inheritance uses and abuses can be verified. These properties make their system attractive for extension to the multiple inheritance case.

We considered EiffelBase, a library of data structures and algorithms which uses multiple inheritance extensively to implement orthogonal features of collections, such as being ordered, or providing random or sequential access, and investigated how interesting uses of multiple inheritance impact reasoning. The main challenges we encountered are: 1) methods and state (fields and data representations) from several parents must be handled, also in the case of diamond

** This work was done while visiting ETH Zurich.

inheritance where a class has common ancestors, and 2) relationships between different views (abstractions) of an object, which abstract predicates capture, must be described and verified. Parent classes are often developed independently, and their views must be reconciled in child classes for the verification of typical code.

After experimenting with Parkinson and Bierman's system, it became clear that extensions were required to accommodate inherited state and enforce relationships between abstractions. We introduce two new specification mechanisms: 1) *export clauses* for static relationships among predicates which apply to one class only, and 2) *axioms* for dynamic relationships which must be preserved by all subclasses. Export clauses and axioms offer two complementary ways to achieve *view shifting* in the logic. View shifting is pervasive and not limited to multiple inheritance: an object can go through a long initialization process after which it is viewed as ready for use, or a method can view an object as read-only. The examples given in this paper illustrate important uses of our extensions. Other uses include abstract class and interface verification, constraining subclass data representations, and view shifting in single inheritance programs.

A gentle warm-up follows in Section 2. Section 3 introduces a programming language with multiple inheritance and Section 4 formalizes its proof system. It is possible to understand the examples in Section 5 at least partially without complete knowledge of the formalization. We discuss limitations of the approach in Section 6, and Section 7 concludes and mentions related work.

2 Warm-up

An apf [4, 3] provides an abstract predicate p for which each class G can define an entry p@G. In Figure 1 on page 3, class COUNTER introduces an apf named Cn which provides an abstraction of integer-valued counters in the logic. It furthermore defines its entry for Cn, namely the Cn@COUNTER predicate. The first argument of an apf predicate is its root object. Since the meaning of an apf predicate depends on the dynamic type of the root object, it can be seen as mirroring dynamic dispatch of object-orientation in the logic. The second argument of an apf predicate or entry is a set of tagged arguments, where tag names provide useful hints about the purpose of tagged values. For example, the tag name cnt describes the value of the counter in Cn apf predicates and entries. Apfs offer high levels of abstraction: apf predicates and entries are treated abstractly by clients, unless information about them is made available in export and axiom clauses as we shall see.

Consider for example class CCELL in Figure 2 on page 4 which inherits from CELL and COUNTER. It combines a mutable integer-valued cell with a counter and increments the count every time a value is stored. Class CCELL introduces apf Cc, which provides an abstraction of such objects in the logic. The val and cnt tags in Cc apf predicates and entries label the stored value and count respectively. The constructor of CCELL yields a Cc apf predicate, as indicated by the postcondition of its Hoare-style specification after the keyword

```
class CELL
define Cell@CELL(x, {val=v}) as x.value ↪ v
feature
    introduce CELL(v: int)
    dynamic {Current.value ↪ _}_{Cell(Current,{val=v})}
    do Current.value := v end

    introduce value(): int
    dynamic {Cell(Current,{val=v})}_{Cell(Current,{val=v}) * Result = v}
    do Result := Current.value end

    introduce set_value(v: int)
    dynamic {Cell(Current,{val=_})}_{Cell(Current,{val=v})}
    do Current.value := v end

    value: int
end

class COUNTER
define Cn@COUNTER(x,{cnt=c}) as x.count ↪ c
feature
    introduce COUNTER()
    dynamic {Current.count ↪ _}_{Cn(Current,{cnt=0})}
    do Current.count := 0 end

    introduce count(): int
    dynamic {Cn(Current,{cnt=c})}_{Cn(Current,{cnt=c}) * Result = c}
    do Result := Current.count end

    introduce increment()
    dynamic {Cn(Current,{cnt=c})}_{Cn(Current,{cnt=c+1})}
    do tmp: int tmp := Current.count Current.count := tmp + 1 end

    count: int
end
```

Fig. 1. The CELL and COUNTER classes.

```
class CCELL inherit CELL COUNTER
define
Cell@CCELL(x,{val=v;cnt=c}) as Cc@CCELL(x,{val=v;cnt=c})
Cn@CCELL(x,{cnt=c}) as Cn@COUNTER(x,{cnt=c})
Cc@CCELL(x,{val=v;cnt=c}) as Cell@CELL(x,{val=v})∗Cn@COUNTER(x,{cnt=c})
export
    ∀x· x : CCELL ⇒ [∀c,v· Cc(x,{val=v;cnt=c})) ⇔ Cell(x,{val=v;cnt=c}) ⇔
    (Cn(x,{cnt=c}) ∗ Rest(x,v))] where { Rest(x,v) = Cell@CELL(x,{val=v})}
feature
    introduce CCELL(v: int) dynamic
    {Current.value ↪ _ ∗ Current.count ↪ _}_{Cc(Current,{val=v;cnt=0})}
    do Precursor{CELL}(v) Precursor{COUNTER}() end

    inherit value(): int dynamic
    {Cc(Current,{val=v;cnt=c})}_{Cc(Current,{val=v;cnt=c}) ∗ Result = v} also
    {Cell(Current,{val=v;cnt=c})}_{Cell(Current,{val=v;cnt=c}) ∗ Result = v}

    override set_value(v: G)
    dynamic {Cc(Current,{val=_;cnt=c})}_{Cc(Current,{val=v;cnt=c+1})}
       also {Cell(Current,{val=_;cnt=c})}_{Cell(Current,{val=v;cnt=c+1})}
    do Current.CCELL::increment() Current.CELL::set_value(v) end

    inherit count(): int dynamic
    {Cc(Current,{val=v;cnt=c})}_{Cc(Current,{val=v;cnt=c}) ∗ Result = c}
    also {Cn(Current,{cnt=c})}_{Cn(Current,{cnt=c}) ∗ Result = c}

    inherit increment()
    dynamic {Cc(Current,{val=v;cnt=c})}_{Cc(Current,{val=v;cnt=c+1})}
       also {Cn(Current,{cnt=c})}_{Cn(Current,{cnt=c+1})}
end

// In an arbitrary class or library:
use_counter(c: COUNTER)
    dynamic {Cn(c,{cnt=v})}_{Cn(c,{cnt=v+10})}

use_cell(c: CELL, v: int)
    dynamic {Cell(c,{val=_})}_{Cell(c,{val=v})}
```

Fig. 2. The CCELL class and two library methods.

dynamic. However, the library routine *use_counter* at the bottom of Figure 2 demands a Cn apf predicate, as it was developed without knowledge of counted cells. A verification problem arises whenever *use_counter* is reused to operate on an instance of CCELL. This is shown in the following proof attempt where assertions and statements are interleaved.

```
{true}
    cc := new CCELL(5)
```

{Cc(cc,{val=5;cnt=0})}
{???}
{Cn(cc,{cnt=0})}
    use_counter(cc)
{Cn(cc,{cnt=10})}
{???}
{Cc(cc,{val=5;cnt=10})}

In order to fill the gaps indicated by ???, we need to relate the abstract Cc and Cn views of a CCELL instance while preserving information about its stored value. If the *use_cell* method is called right after the call to *use_counter*, a view shift to obtain a Cell apf predicate is similarly needed. Completing the proof becomes easy with the formal system defined in Section 4.

3 A programming language with specifications

3.1 Syntax

The grammar of our kernel language with multiple inheritance and specifications is shown in Figure 3. A sequence of c's is denoted by $\bar{c}$. The letters G and H are used for class names, p for apf names, t for tag names, w for ordinary predicate names, a for axiom names, m for method names, and f for field names. Variables are denoted by u, x, y and z.

A definition L of class G is divided into different sections: **inherit**, **define**, **export**, **axiom** and **feature**. Define clauses are written in the **define** section and give the apf entries for G. Export and axiom clauses provide additional information which can be used for reasoning. They can be used to specify view shifts, in particular, as will be shown later. Method and field declarations are contained in the **feature** section of a class.

Separate namespaces exist for class names, p, w, a, m and f. The type system ensures absence of clashes when names are introduced. This precludes method overloading and field shadowing, for instance, and guarantees that methods or fields with the same name in parent classes stem from common ancestors.

Call statements of the form x := y.G::m($\bar{z}$) and y.G::($\bar{z}$) denote direct method calls in C++ style. Such calls follow static dispatch and can be used to call ancestor versions of a method or constructors in parent classes, for example.

Constructors are not treated the same as in [3], where the parent class constructor is implicitly called. A constructor here is simply an introduced method m where m is a class name. Except for the restriction that subclasses cannot inherit or override constructors, no special treatment is needed otherwise.

To provide subclasses with the opportunity to respecify a method and to simplify the proof rules that follow later, we require a subclass to inherit or override explicitly all non-constructor methods present in its parents. The shared semantics of multiple inheritance is used, which is popular in Eiffel and known as 'virtual' inheritance in C++. If field f is listed in multiple parents of G, a single field f will be available in G. If a method is overridden, all ancestor versions of

L ::= Ab **class** G **inherit** $\overline{\text{H}}$ **define** $\overline{\text{D}}$ **export** $\overline{\text{E}}$ **axiom** $\overline{\text{A}}$ **feature** $\overline{\text{M}}$ $\overline{\text{F}}$ **end**	
Ab ::= **abstract** | ϵ	
D ::= p@G(x,{$\overline{\text{t} = \text{y}}$}) **as** P	*Define clause*
E ::= P **where** {$\overline{\text{W}}$}	*Export clause*
W ::= w($\overline{\text{x}}$) = P	*Where clause*
A ::= a: P	*Axiom clause*
M ::= **introduce** m(Args) Rt Sd Ss B	*Method declaration*
| **override** m(Args) Rt Sd Ss B	
| **inherit** m(Args) Rt Sd Ss	
| **introduce abstract** m(Args) Rt Sd	
| **inherit abstract** m(Args) Rt Sd	
F ::= f: Type	*Field declaration*
Sd ::= **dynamic** Spec	*Dynamic specification*
Ss ::= **static** Spec	*Static specification*
Spec ::= {P}_{Q} | {P}_{Q} **also** Spec	*Specification*
B ::= **do** $\overline{\text{s}}$ **end**	*Method body*
s ::= x: Type	*Local variable declaration*
| x := e	*Assignment*
| x := y.f	*Field lookup*
| x.f := e	*Field assignment*
| x := y.m($\overline{\text{z}}$) | y.m($\overline{\text{z}}$)	*Dynamically dispatched call*
| x := y.G::m($\overline{\text{z}}$) | y.G::m($\overline{\text{z}}$)	*Direct method call*
| x := **new** G	*Object allocation*
e ::= x | e + e | e = e | **Void** | 0 | 1 | 2 | ...	*Expression*
Type ::= int | bool | G	
Args ::= $\overline{\text{x: Type}}$	*Formal arguments*
Rt ::= ϵ | : Type	*Return type*

Fig. 3. The kernel language grammar.

it are overridden. To avoid ambiguity, a class can inherit a method only if its implementation (if there is one) is the same along all inheritance paths. Direct method calls can encode language mechanisms which allow a particular ancestor implementation to be chosen, so no generality is lost.

We assume the formal argument names of methods stay the same in subclasses. This simplifies the proof rules that follow, which would otherwise need additional substitutions.

Void corresponds to 'null' in other languages. Two reserved program variables **Current** and **Result** denote the current object ('this') and the result of a function call respectively. **Current** is never **Void**.

3.2 Operational semantics

Restrictions imposed by the type system together with the shared semantics of multiple inheritance ensure the absence of ambiguity in field and method lookup.

The shared semantics also ensures that 1) only dynamic type information is needed at runtime (cf. what 'select' clauses of Eiffel demand), and 2) the usual semantics of casts can be adopted (cf. C++, where the cast expression and the casted expression might differ in value [5]). Point 2 is already reflected in the grammar, where assignment can be performed without explicit upcasting.

The operational semantics is standard and omitted. Configurations contain a stack, a heap and a sequence of statements under execution. The stack maps variables to values which include object ids. The heap maps object ids to records containing a dynamic type G and field-value mappings.

4 Formalization of the proof system

An extension of the proof system in [3] is described here. For space reasons we focus mostly on the new extensions. Details of the formal system are best understood by studying the examples in Section 5.

4.1 Logic syntax and semantics

The predicates used in specifications and proofs have the following grammar.

$$\begin{aligned} P, Q, S, T, \Delta ::= {} & \forall x{\cdot}P \mid P \Rightarrow Q \mid \mathrm{false} \mid e_1 = e_2 \mid x : G \mid x <: G \mid x.f \hookrightarrow e \mid P * Q \\ & \mid p(x,\{\overline{t = e}\}) && \textit{Apf predicate} \\ & \mid p@G(x,\{\overline{t = e}\}) && \textit{Apf entry} \\ & \mid w(\overline{x}) && \textit{Ordinary predicate} \end{aligned}$$

The predicate x : G means x references an object whose dynamic type is exactly G, and x <: G means x references an object whose dynamic type is a subtype of G. In both cases $x \neq$ **Void**, and $x : G \Rightarrow x <: G$ holds. Within a context, if x is declared of type G then x <: G whenever $x \neq$ **Void**.

The notation used for apf predicates and their entries for particular classes comes from jStar [6]. The second argument is a set of tagged expressions where each tag (denoted by t) can occur at most once. Two such sets are equal if they use the same tags and associate equal expressions with equal tags. Bundling several arguments of a predicate into a tagged set provides order-independence, which is especially useful for multiple inheritance. Each class can provide definitions of its apf entries in the **define** section. An informal discussion of apfs was given in Section 2. For their formal semantics the reader is referred to [3] for lack of space. The root object of an apf predicate is never **Void**.

Other predicates have the usual intuitionistic separation logic semantics. Informally the predicate $x.f \hookrightarrow e$ means that the f field of object x has value e, and $P * Q$ means that P and Q hold for disjoint portions of the heap. Readers are referred to [7–9] for a formal treatment of separation logic. Symbols such as $\Leftrightarrow$, $\neg$, true, $\vee$, $\wedge$ and $\exists$ are encoded in the standard way. Every occurrence of _ in a predicate denotes a fresh existentially quantified variable, where the quantifier is placed in the innermost position. $FV(P)$ denotes the free variables of P; every method precondition P must satisfy **Result** $\notin FV(P)$.

In the rest of the paper the symbols P, Q, S and T are used for assertions and predicates, and Δ for assumptions.

4.2 Specification refinement

Parkinson and Bierman introduced the notion of specification refinement in [3] to formalize behavioral subtyping. If the specification $\{P_1\}\_\{Q_1\}$ is refined by $\{P_2\}\_\{Q_2\}$, then any $\bar{s}$ which satisfies $\{P_1\}\_\{Q_1\}$ also satisfies $\{P_2\}\_\{Q_2\}$. If this is the case we write $\Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$, which denotes the existence of a proof tree with leaves $\Delta \vdash \{P_1\}\_\{Q_1\}$ and root $\Delta \vdash \{P_2\}\_\{Q_2\}$ built with the structural rules of separation logic (Consequence, Frame, Auxiliary Variable Elimination, Disjunction, and others). In the context of method specification refinement, the Frame and Consequence rules are given by:

$$\frac{\Delta \vdash \{P\}\_\{Q\}}{\Delta \vdash \{P * T\}\_\{Q * T\}}\text{Frame} \qquad \frac{\Delta \Rightarrow (P' \Rightarrow P)\ \Delta \vdash \{P\}\_\{Q\}\ \Delta \Rightarrow (Q \Rightarrow Q')}{\Delta \vdash \{P'\}\_\{Q'\}}\text{Conseq}$$

provided $\mathbf{Result} \notin FV(T)$.

The Frame rule expresses that disjoint potions of the heap stay unchanged.

Method specifications can be combined with **also** (Definition 1 in [3]):

$$\{P_1\}\_\{Q_1\}\ \mathbf{also}\ \{P_2\}\_\{Q_2\} \stackrel{\text{def}}{=} \{(P_1 \wedge x = 1) \vee (P_2 \wedge x \neq 1)\}\_\{(Q_1 \wedge x = 1) \vee (Q_2 \wedge x \neq 1)\}$$

where x denotes a fresh auxiliary variable. The specifications $\{P_1\}\_\{Q_1\}$ and $\{P_2\}\_\{Q_2\}$ are *equivalent w.r.t.* Δ iff both $\Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$ and $\Delta \vdash \{P_2\}\_\{Q_2\} \Longrightarrow \{P_1\}\_\{Q_1\}$. Two specifications are *equivalent* iff they are equivalent w.r.t. all Δ. It can be shown that **also** is commutative, associative and idempotent modulo equivalence with identity $\{\text{false}\}\_\{\text{true}\}$. The notation $\mathbf{also}_{i \in I}\ \{P_i\}\_\{Q_i\}$ abbreviates $\{P_{e_1}\}\_\{Q_{e_1}\}\ \mathbf{also} \ldots \mathbf{also}\ \{P_{e_m}\}\_\{Q_{e_m}\}$, where $e_1 \ldots e_m$ are the elements of set I. Furthermore, when I is the empty set:

$$\mathbf{also}_{i \in \emptyset}\ \{P_i\}\_\{Q_i\} \stackrel{\text{def}}{=} \{\text{false}\}\_\{\text{true}\}$$

It always holds that $\Delta \vdash \{P\}\_\{Q\} \Longrightarrow \{\text{false}\}\_\{\text{true}\}$. Other useful lemmas involving **also** are given in Section 4.9. Finally, following [3],

$$\Delta \vdash \{P_1\}\_\{Q_1\} \stackrel{\mathbf{Current}:G}{\Longrightarrow} \{P_2\}\_\{Q_2\} \stackrel{\text{def}}{=} \Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2 * \mathbf{Current} : G\}\_\{Q_2\}.$$

4.3 The specification environment

Most of the proof rules which follow use an environment Γ, which maps axiom and method names to their specifications for all classes in a program:

$\Gamma ::= G.a \mapsto P$	*Axiom specification*
$\mid G.m \mapsto (\overline{x}, \{P\}\_\{Q\})$	*Method dynamic specification*
$\mid G::m \mapsto (\overline{x}, \{S\}\_\{T\})$	*Method static specification*
$\mid \overline{\Gamma}$	

The $\overline{x}$ in a specification of m denote its formal argument names. Γ is guaranteed to be a partial function for well-typed programs, and we write $\Gamma(G.a) = P$ for $G.a \mapsto P \in \Gamma$, etc.

4.4 Export information verification

A class can make information about itself available to other classes in an export clause. Export clauses are frequently used to specify view shifts in terms of relationships between apfs or their entries, and to expose apf entry definitions. Information can also be hidden in predicates defined after the keyword **where**: the definitions are not exported, so other classes must treat these predicates abstractly.

Export information must be verified since other classes use it for reasoning. Under the predicate definitions following **where**, the assumptions about a class must imply exported information:

$$\frac{[\Delta \wedge (\forall \overline{x_1} \cdot w_1(\overline{x_1}) \Leftrightarrow Q_1) \wedge \ldots \wedge (\forall \overline{x_n} \cdot w_n(\overline{x_n}) \Leftrightarrow Q_n)] \Rightarrow P \qquad \text{(Validity)}}{\Delta \vdash_e P \textbf{ where } \{w_1(\overline{x_1}) = Q_1; \ldots; w_n(\overline{x_n}) = Q_n\}}$$

In Figure 2, for example, class CCELL exports the equivalence of Cc and Cell apf predicates whenever the root object has dynamic type CCELL.

4.5 Axiom verification

Information about a class and all its subclasses can be made available in an axiom clause. This knowledge can be used later to verify method bodies.

In the rules for axiom verification, the assumptions Δ include information about class G and export information from all other classes. A subclass must preserve all axioms of its parents and may refine the predicate associated with an axiom name (see the Parent consistency proof obligation). A non-abstract class must also show that the predicate holds for its instances (the Implication obligation).

$$\frac{\begin{array}{l}\forall i \in I \cdot \Gamma(H_i.a) = Q_i \ \wedge \ \forall j \in (1..n \setminus I) \cdot H_j.a \notin \mathit{dom}(\Gamma) \\ (\Delta \wedge P) \Rightarrow \bigwedge_{i \in I} Q_i \qquad \text{(Parent consistency)} \\ (\Delta \wedge \textbf{Current} : G) \Rightarrow P \qquad \text{(Implication)}\end{array}}{\Delta; \Gamma \vdash_a a\text{: } P \text{ in } G \text{ parents } H_1 \ldots H_n}$$

$$\frac{\begin{array}{l}\forall i \in I \cdot \Gamma(H_i.a) = Q_i \ \wedge \ \forall j \in (1..n \setminus I) \cdot H_j.a \notin \mathit{dom}(\Gamma) \\ (\Delta \wedge P) \Rightarrow \bigwedge_{i \in I} Q_i \qquad \text{(Parent consistency)}\end{array}}{\Delta; \Gamma \vdash_a a\text{: } P \text{ in } \textbf{abstract } G \text{ parents } H_1 \ldots H_n}$$

The diamond inheritance example in Section 5.3 uses axioms extensively.

4.6 Statement verification

The assumptions Δ used to verify statements contain information about the enclosing class as well as export and axiom information from all other classes. The rules for most statements are standard (see e.g. [9, 3]). For allocation:

$$\frac{\mathit{allfields}(G) = \{f_1, f_2, \ldots, f_n\}}{\Delta; \Gamma \vdash_s \{\text{true}\} x := \textbf{new } G\{x.f_1 \hookrightarrow \_ * x.f_2 \hookrightarrow \_ * \ldots * x.f_n \hookrightarrow \_ * x : G\}}$$

where *allfields*(G) denotes the set of field names listed in G and all its ancestors.

Dynamically dispatched calls use the dynamic specs of methods in Γ, while direct calls use the static ones. Provided x is not y and x is not in $\overline{z}$, the rules for result-returning calls are:

$$\frac{\Gamma(\mathrm{G.m}) = (\overline{u},\{\mathrm{P}\}\text{-}\{\mathrm{Q}\})}{\begin{array}{c}\Delta;\Gamma \vdash_s \{\mathrm{P}[\mathrm{y},\overline{z}/\mathbf{Current},\overline{u}] * \mathrm{y} <: \mathrm{G}\} \\ \mathrm{x} := \mathrm{y.m}(\overline{z}) \\ \{\mathrm{Q}[\mathrm{y},\overline{z},\mathrm{x}/\mathbf{Current},\overline{u},\mathbf{Result}]\}\end{array}}$$

$$\frac{\Gamma(\mathrm{G::m}) = (\overline{u},\{\mathrm{S}\}\text{-}\{\mathrm{T}\})}{\begin{array}{c}\Delta;\Gamma \vdash_s \{\mathrm{S}[\mathrm{y},\overline{z}/\mathbf{Current},\overline{u}] * \mathrm{y} \neq \mathbf{Void}\} \\ \mathrm{x} := \mathrm{y.G::m}(\overline{z}) \\ \{\mathrm{T}[\mathrm{y},\overline{z},\mathrm{x}/\mathbf{Current},\overline{u},\mathbf{Result}]\}\end{array}}$$

Two important structural rules here are Frame and Consequence. The Frame rule is the key to local reasoning. Provided $\overline{s}$ modifies no variable in $FV(\mathrm{T})$:

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\overline{s}\{\mathrm{Q}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P} * \mathrm{T}\}\overline{s}\{\mathrm{Q} * \mathrm{T}\}}\,\text{Frame}$$

The rule of Consequence allows the use of assumptions Δ:

$$\frac{\Delta \Rightarrow (\mathrm{P}' \Rightarrow \mathrm{P}) \qquad \Delta;\Gamma \vdash_s \{\mathrm{P}\}\overline{s}\{\mathrm{Q}\} \qquad \Delta \Rightarrow (\mathrm{Q} \Rightarrow \mathrm{Q}')}{\Delta;\Gamma \vdash_s \{\mathrm{P}'\}\overline{s}\{\mathrm{Q}'\}}\,\text{Conseq}$$

4.7 Method verification

The rules for method verification in [3] are extended here to the multiple inheritance case. As for statement verification, the assumptions Δ used to verify method definitions contain information about the method's enclosing class as well as export and axiom information from all other classes.

The rule for method introduction requires no modification for multiple inheritance. A newly introduced method's static and dynamic specifications must be consistent (the Dynamic dispatch proof obligation), and its body must satisfy the static specification (the Body verification obligation).

$$\frac{\begin{array}{l}\mathrm{B} = \mathbf{do}\ \overline{s}\ \mathbf{end} \\ \mathrm{Sd} = \mathbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \mathbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\mathbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \qquad \text{(Dynamic dispatch)} \\ \Delta;\Gamma \vdash_s \{\mathrm{S_G}\}\overline{s}\{\mathrm{T_G}\} \qquad \text{(Body verification)}\end{array}}{\Delta;\Gamma \vdash_m \mathbf{introduce}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ G\ parents}\ \overline{\mathrm{H}}}$$

An abstract method can be introduced without any proof obligations, since there is only a dynamic specification and no method body.

$$\frac{}{\Delta;\Gamma \vdash_m \mathbf{introduce\ abstract}\ \mathrm{m(Args)\ Rt\ Sd\ in\ G\ parents}\ \overline{\mathrm{H}}}$$

The next rule is used whenever an abstract method is implemented or a method body is redefined. Consistency must be proven between the new dynamic specification and those in parent classes (the Behavioral subtyping proof obligation). The other proof obligations are identical to those for method introduction above. The $H_1 \ldots H_n$ are the immediate superclasses of G.

$$\frac{\begin{array}{ll}\forall i \in I \cdot \Gamma(H_i.m) = (\overline{x},\{P_{Hi}\}\text{-}\{Q_{Hi}\}) \ \wedge \ \forall j \in (1..n \setminus I) \cdot H_j.m \notin dom(\Gamma) & \\ B = \textbf{do } \overline{s} \textbf{ end} & \\ Sd = \textbf{dynamic } \{P_G\}\text{-}\{Q_G\} & \\ Ss = \textbf{static } \{S_G\}\text{-}\{T_G\} & \\ \Delta \vdash \{P_G\}\text{-}\{Q_G\} \Longrightarrow (\textbf{also}_{i \in I}\ \{P_{Hi}\}\text{-}\{Q_{Hi}\}) & \text{(Behavioral subtyping)} \\ \Delta \vdash \{S_G\}\text{-}\{T_G\} \overset{\textbf{Current}:G}{\Longrightarrow} \{P_G\}\text{-}\{Q_G\} & \text{(Dynamic dispatch)} \\ \Delta;\Gamma \vdash_s \{S_G\}\overline{s}\{T_G\} & \text{(Body verification)}\end{array}}{\Delta;\Gamma \vdash_m \textbf{override } m(Args)\ Rt\ Sd\ Ss\ B \text{ in } G \text{ parents } H_1 \ldots H_n}$$

When a non-abstract method is inherited, its static specification must follow from those in parents (the Inheritance obligation ensures this). The Behavioral subtyping and Dynamic dispatch obligations serve the same purposes as mentioned before.

$$\frac{\begin{array}{ll}\forall i \in I \cdot \Gamma(H_i.m) = (\overline{x},\{P_{Hi}\}\text{-}\{Q_{Hi}\}) \ \wedge \ \forall k \in (1..n \setminus I) \cdot H_k.m \notin dom(\Gamma) & \\ \forall j \in J \cdot \Gamma(H_j::m) = (\overline{x},\{S_{Hj}\}\text{-}\{T_{Hj}\}) \ \wedge \ \forall l \in (1..n \setminus J) \cdot H_l::m \notin dom(\Gamma) & \\ Sd = \textbf{dynamic } \{P_G\}\text{-}\{Q_G\} & \\ Ss = \textbf{static } \{S_G\}\text{-}\{T_G\} & \\ \Delta \vdash \{P_G\}\text{-}\{Q_G\} \Longrightarrow (\textbf{also}_{i \in I}\ \{P_{Hi}\}\text{-}\{Q_{Hi}\}) & \text{(Behavioral subtyping)} \\ \Delta \vdash (\textbf{also}_{j \in J}\ \{S_{Hj}\}\text{-}\{T_{Hj}\}) \Longrightarrow \{S_G\}\text{-}\{T_G\} & \text{(Inheritance)} \\ \Delta \vdash \{S_G\}\text{-}\{T_G\} \overset{\textbf{Current}:G}{\Longrightarrow} \{P_G\}\text{-}\{Q_G\} & \text{(Dynamic dispatch)}\end{array}}{\Delta;\Gamma \vdash_m \textbf{inherit } m(Args)\ Rt\ Sd\ Ss \text{ in } G \text{ parents } H_1 \ldots H_n}$$

The next rule applies whenever an abstract method is inherited or a non-abstract method is inherited and made abstract. Such a method has no static specification, so only the consistency of its dynamic specification w.r.t those in parent classes is required with the Behavioral subtyping proof obligation.

$$\frac{\begin{array}{ll}\forall i \in I \cdot \Gamma(H_i.m) = (\overline{x},\{P_{Hi}\}\text{-}\{Q_{Hi}\}) \ \wedge \ \forall j \in (1..n \setminus I) \cdot H_j.m \notin dom(\Gamma) & \\ Sd = \textbf{dynamic } \{P_G\}\text{-}\{Q_G\} & \\ \Delta \vdash \{P_G\}\text{-}\{Q_G\} \Longrightarrow (\textbf{also}_{i \in I}\ \{P_{Hi}\}\text{-}\{Q_{Hi}\}) & \text{(Behavioral subtyping)}\end{array}}{\Delta;\Gamma \vdash_m \textbf{inherit abstract } m(Args)\ Rt\ Sd \text{ in } G \text{ parents } H_1 \ldots H_n}$$

4.8 Class and program verification

Consider the following two verification rules. For class verification, different assumptions are used to verify the various class sections. The formula Δ_{APF} contains class-specific information and is used to verify export clauses. The assumptions Δ_E contain export information from all classes, and are used together with Δ_{APF} to verify axioms. The formula Δ_A contains axiom information of all classes, and is used with Δ_{APF} and Δ_E in method definition verification.

$$\frac{\forall \mathrm{E_i} \in \overline{\mathrm{E}} \cdot \Delta_{APF} \vdash_e \mathrm{E_i} \quad \forall \mathrm{A_i} \in \overline{\mathrm{A}} \cdot (\Delta_{APF} \wedge \Delta_E); \Gamma \vdash_a \mathrm{A_i} \text{ in Ab G parents } \overline{\mathrm{H}} \quad \forall \mathrm{M_i} \in \overline{\mathrm{M}} \cdot (\Delta_{APF} \wedge \Delta_E \wedge \Delta_A); \Gamma \vdash_m \mathrm{M_i} \text{ in G parents } \overline{\mathrm{H}}}{\Delta_{APF}, \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{Ab}\ \mathbf{class}\ \mathrm{G}\ \mathbf{inherit}\ \overline{\mathrm{H}}\ \mathbf{define}\ \overline{\mathrm{D}}\ \mathbf{export}\ \overline{\mathrm{E}}\ \mathbf{axiom}\ \overline{\mathrm{A}}\ \mathbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}}\ \mathbf{end}}$$

Finally, here is the rule for program verification:

$$\frac{\begin{array}{l}\forall \mathrm{i} \in 1..\mathrm{n} \cdot \mathrm{L_i} = \ldots \mathbf{class}\ \mathrm{G_i} \ldots \mathbf{export}\ \overline{\mathrm{E_i}}\ \mathbf{axiom}\ \overline{\mathrm{A_i}}\ \mathbf{feature} \ldots \mathbf{end} \\ \Delta_E = \bigwedge_{\mathrm{i} \in 1..\mathrm{n}} \bigwedge_{\mathrm{E_{i_k}} \in \overline{\mathrm{E_i}}} \mathit{exportinfo}(\mathrm{E_{i_k}}) \\ \Delta_A = \bigwedge_{\mathrm{i} \in 1..\mathrm{n}} \bigwedge_{\mathrm{A_{i_k}} \in \overline{\mathrm{A_i}}} \mathit{axiominfo}(\mathrm{G_i}, \mathrm{A_{i_k}}) \\ \Gamma = \mathit{specs}(\mathrm{L_1} \ldots \mathrm{L_n}) \\ \forall \mathrm{i} \in 1..\mathrm{n} \cdot \ \mathit{apf}(\mathrm{L_i}),\ \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L_i} \\ \Delta_E \wedge \Delta_A; \Gamma \vdash_s \{\mathrm{true}\}\overline{\mathrm{s}}\{\mathrm{true}\}\end{array}}{\vdash_p \mathrm{L_1} \ldots \mathrm{L_n}\ \overline{\mathrm{s}}}$$

$\mathit{exportinfo}(\mathrm{P}\ \mathbf{where} \ldots) \stackrel{\text{def}}{=} \mathrm{P}$
$\mathit{axiominfo}(\mathrm{G},\ \mathrm{a}\colon \mathrm{P}) \stackrel{\text{def}}{=} \forall \mathrm{x} <\colon \mathrm{G} \cdot \mathrm{P}[\mathrm{x}/\mathbf{Current}]$, where x is fresh.

Predicate definitions following the **where** keyword are hidden by *exportinfo*, and the definition of *axiominfo* reflects the fact that subclasses preserve axioms.

The function *apf* translates the abstract predicate family definitions of a class into a formula. It is adapted from [3] for tagged predicate arguments:

$$\begin{array}{l}\mathit{apf}(\mathrm{Ab}\ \mathbf{class}\ \mathrm{G} \ldots \mathbf{define}\ \mathrm{D_1}\ \mathrm{D_2} \ldots \mathrm{D_n}\ \mathbf{export} \ldots \mathbf{end}) \\ \qquad \stackrel{\text{def}}{=} \mathit{apf}_{\mathrm{G}}(\mathrm{D_1}) \wedge \ldots \wedge \mathit{apf}_{\mathrm{G}}(\mathrm{D_n}) \\ \mathit{apf}_{\mathrm{G}}(\mathrm{p@G(x,Y)}\ \mathbf{as}\ \mathrm{P}) \\ \qquad \stackrel{\text{def}}{=} \mathit{FtoE}(\mathrm{p,G,Y}) \wedge \mathit{EtoD}(\mathrm{p@G(x,Y)}\ \mathbf{as}\ \mathrm{P}) \wedge (\forall \mathrm{x} <\colon \mathrm{G} \cdot \mathit{TR}(\mathrm{p,x,Y})) \\ \mathit{FtoE}(\mathrm{p,G,}\{\overline{\mathrm{t} = \mathrm{y}}\}) \\ \qquad \stackrel{\text{def}}{=} \forall \mathrm{x}, \overline{\mathrm{y}} \cdot \mathrm{x} : \mathrm{G} \Rightarrow [\mathrm{p(x,}\{\overline{\mathrm{t} = \mathrm{y}}\}) \Leftrightarrow \mathrm{p@G(x,}\{\overline{\mathrm{t} = \mathrm{y}}\})] \\ \mathit{EtoD}(\mathrm{p@G(x,}\{\overline{\mathrm{t} = \mathrm{y}}\})\ \mathbf{as}\ \mathrm{P}) \\ \qquad \stackrel{\text{def}}{=} \forall \mathrm{x}, \overline{\mathrm{y}} \cdot \mathrm{p@G(x,}\{\overline{\mathrm{t} = \mathrm{y}}\}) \Leftrightarrow \mathrm{P} \\ \mathit{TR}(\mathrm{p,x,}\{\overline{\mathrm{t} = \mathrm{y}}\}) \\ \qquad \stackrel{\text{def}}{=} \bigwedge_{\overline{\mathrm{t'=y'}} + \overline{\mathrm{t''=y''}} \equiv \overline{\mathrm{t=y}}} \forall \overline{\mathrm{y'}} \cdot \mathrm{p(x,}\{\overline{\mathrm{t' = y'}}\}) \Leftrightarrow \mathrm{p(x,}\{\overline{\mathrm{t' = y'}}\ +\ \overline{\mathrm{t'' = \_}}\})\end{array}$$

Theorem. The program verification rule is sound. (The proof depends on the layered assumption structure which avoids circularity in reasoning.)

4.9 Useful lemmas

Lemmas 1 and 2 are frequently used in proofs of Behavioral Subtyping and Inheritance:

Lemma 1. $\Delta \vdash (\mathbf{also}_{\mathrm{i} \in \mathrm{I}}\ \{\mathrm{P_i}\}\_\{\mathrm{Q_i}\}) \Longrightarrow \{\mathrm{P_k}\}\_\{\mathrm{Q_k}\}$ for all $\mathrm{k} \in \mathrm{I}$.

Lemma 2. If $\Delta \vdash \{P\}\_\{Q\} \Longrightarrow \{S_i\}\_\{T_i\}$ for all $i \in I$, then $\Delta \vdash \{P\}\_\{Q\} \Longrightarrow (\mathbf{also}_{i\in I}\ \{S_i\}\_\{T_i\})$.

For Body Verification:

Lemma 3. If $\Delta;\Gamma \vdash_s \{S_i\}\bar{s}\{T_i\}$ for all $i \in I$, then under assumptions Δ and Γ, $\bar{s}$ satisfies $(\mathbf{also}_{i\in I}\ \{S_i\}\_\{T_i\})$.

5 Examples

5.1 Syntactic conventions

We introduce syntactic sugar to enhance readability and reduce space overhead. Empty sections in classes are simply omitted. The statement x := **new** G($\overline{y}$) abbreviates x := **new** G followed by x.G($\overline{y}$), and **Precursor**{G}($\overline{x}$) denotes the call **Current**.G::G($\overline{x}$). An axiom clause copied verbatim from a parent is simply left out. The notation **dynstat** Spec abbreviates **dynamic** Spec and **static** Spec. Following [3], we use the inductive syntactic function $\langle\!\langle P \rangle\!\rangle_G$ to derive static from dynamic specifications, whose only interesting case is

$$\langle\!\langle p(\mathbf{Current}, \{\overline{t = e}\}) \rangle\!\rangle_G \stackrel{\text{def}}{=} p@G(\mathbf{Current}, \{\overline{t = e}\})$$

If only the dynamic specification $\{P\}\_\{Q\}$ is given for a non-abstract method in class G, its static specification is assumed to be $\{\langle\!\langle P \rangle\!\rangle_G\}\_\{\langle\!\langle Q \rangle\!\rangle_G\}$. The dynamic dispatch obligation is then trivially satisfied:

Lemma 4 (Lemma 5 from [3])**.** $apf(L) \vdash \{\langle\!\langle P \rangle\!\rangle_G\}\_\{\langle\!\langle Q \rangle\!\rangle_G\} \overset{\mathbf{Current}:G}{\Longrightarrow} \{P\}\_\{Q\}$, where L is class G's definition.

Suppose H lists M=...C... where C=Ab m(Args) Rt Sd and m≠H. Then if G inherits from H and omits m it is assumed to list **inherit** C. The static specification of m in G is then derived as usual with $\langle\!\langle P \rangle\!\rangle_G$ if $Ab \neq \epsilon$.

The examples assume that every class implicitly exports tag reduction information. In other words, for every entry (p@G(x,Y) **as** P) in the **define** section of a class G, ($\forall$x <: G · *TR*(p,x,Y) **where** {}) is implicitly exported.

5.2 Intertwining ancestor functionality

Classes CELL and COUNTER are shown in Figure 1. CELL models mutable integer-valued cells and introduces apf Cell, while COUNTER introduces apf Cn. Class CCELL in Figure 2 inherits from both and overrides *set_value* to store the value and increment the count. CCELL introduces apf Cc to provide an abstraction of such objects in the logic. The single export clause is easy to verify. For the constructor we have to prove Body Verification:

{**Current**.value ↪ _ ∗ **Current**.count ↪ _}
  Current.CELL::CELL(v)
{Cell@CELL(**Current**,{val=v}) ∗ **Current**.count ↪ _}
  Precursor{COUNTER}()
{Cell@CELL(**Current**,{val=v}) ∗ Cn@COUNTER(**Current**,{cnt=0})}
{Cc@CCELL(**Current**,{val=v;cnt=0})}

The constructor body simply passes the needed attributes to parent constructors and treats their internal representations abstractly thereafter.

For *value*, Inheritance is proved by Lemma 2: an application of the Frame rule (with C@COUNTER(**Current**,{cnt=c})) and then Consequence derives each **also**-ed static spec in CCELL. In the proof of Behavioral Subtyping, we 'choose' the Cell dynamic spec with Lemma 1 and perform tag reduction with Auxiliary Variable Elimination and Consequence.

Behavioral Subtyping of *set_value* is similar and its Body Verification proceeds as follows:

{Cell@CCELL(**Current**,{val=_;cnt=c})}
  Current.CCELL::increment()
{Cell@CCELL(**Current**,{val=_;cnt=c+1})}
{Cell@CELL(**Current**,{val=_}) ∗ Cn@COUNTER(**Current**,{cnt=c+1})}
  Current.CELL::set_value(v)
{Cell@CCELL(**Current**,{val=v;cnt=c+1})}

An application of Consequence proves the other **also**-ed static spec, and Lemma 3 combines them. Note that Cell@CCELL must be 'grown' to include the state parcels Cell@CELL and Cn@COUNTER because *set_value* operates on both.

Now consider the two library routines at the bottom of Figure 2. The export clause enables the necessary view shifts to prove:

{true} cc := **new** CCELL(5) use_counter(cc) {Cc(cc,{val=5;cnt=10})}
{true} cc := **new** CCELL(5) use_cell(cc,20) {Cc(cc,{val=20;cnt=_})}

Information about cnt is lost in the second triple's postcondition, which is sound because *use_cell* can potentially call *set_value* many times. In a version of CCELL where Cn@CCELL is defined to include the Cell@CELL state and the equivalence of Cc, Cn and Cell is exported, information about val will likewise be lost in the first triple.

Note that dynamic type information is required for the **export**-based view shifts: CCELL does not oblige its subclasses to provide them. We will see later that clients can perform **axiom**-based view shifts with only static type information, but then subclasses are all constrained.

The proof system can enforce interesting access control patterns in verified programs. Consider class CCEL2 in Figure 4 which has the same executable code as CCELL but different specifications. It specifies a one-directional view shift from Cell to Cn in an export clause. The constructor produces a Cell apf predicate with which methods *value*, *set_value* and *count* can be called. Verified clients cannot call *increment* with the Cell view. They must shift to the Cn view, yet they lack information to shift back after the call: no export or axiom clause is available to do this, and every method producing a Cell predicate requires one. So cnt in the Cell view reflects precisely how many times the value has been set. Once *increment* is called, verified clients can never regain the needed capability to call *value* and *set_value*. Abstract predicates, view shifts and method specifications can enforce complex protocols in verified code.

class CCEL2 **inherit** CELL COUNTER
define
Cell@CCEL2(x,{val=v;cnt=c}) **as** Cell@CELL(x,{val=v})∗Cn@COUNTER(x,{cnt=c})
Cn@CCEL2(x,{cnt=c}) **as** Cn@COUNTER(x,{cnt=c})
export
  ∀x· x : CCEL2 ⇒ [∀v,c· Cell(x,{val=v;cnt=c}) ⇒ Cn(x,{cnt=c})] **where** {}
feature
  introduce CCEL2(v: int) **dynamic**
  {**Current**.value ↪ _ ∗ **Current**.count ↪ _}_{Cell(**Current**,{val=v;cnt=0})}
  do Precursor{CELL}(v) **Precursor**{COUNTER}() **end**

  inherit value(): int **dynamic**
  {Cell(**Current**,{val=v;cnt=c})}_{Cell(**Current**,{val=v;cnt=c}) ∗ **Result** = v}

  override set_value(v: G)
  dynamic {Cell(**Current**,{val=_;cnt=c})}_{Cell(**Current**,{val=v;cnt=c+1})}
  do Current.CCEL2::increment() **Current**.CELL::set_value(v) **end**

  inherit count(): int **dynamic**
  {Cell(**Current**,{val=v;cnt=c})}_{Cell(**Current**,{val=v;cnt=c}) ∗ **Result** = c}
  also {Cn(**Current**,{cnt=c})}_{Cn(**Current**,{cnt=c}) ∗ **Result** = c}
end

Fig. 4. The CCEL2 class.

5.3 Diamond inheritance

Verification of multiple inheritance requires proper handling of state from several parent classes. Diamond inheritance complicates matters because common ancestor state is shared. This is unproblematic for our proof system, although abstraction of the shared state is typically lost. Diamond inheritance can moreover require much view shifting, which this example achieves with axiom clauses.

Consider in Figure 5 classes PERSON and STUDENT, which introduce apfs P and S respectively. View shifts between S and P are specified in the **axiom** section of STUDENT. Class MUSICIAN (not shown) is similar to STUDENT. It introduces apf M and specifies axiom-based view shifts between M and P. In Figure 6 class SMUSICIAN inherits from STUDENT and MUSICIAN. It introduces apf SM and more view shifts. A diamond is formed with PERSON at the top, and the state from PERSON is shared in SMUSICIAN.

The Implication proof of axiom SM_S uses export information from STUDENT and MUSICIAN. Under the assumption **Current** : SMUSICIAN

SM@SMUSICIAN(**Current**,{age=a;exm=e;pfm=p})
⇔ P@PERSON(**Current**,{age=a}) ∗ RestStoP@STUDENT(**Current**,{exm=e}) ∗
  RestMtoP@MUSICIAN(**Current**,{pfm=p})
⇔ S@STUDENT(**Current**,{age=a;exm=e}) ∗
  RestMtoP@MUSICIAN(**Current**,{pfm=p})

$\Leftrightarrow$ S@SMUSICIAN(**Current**,{age=a;exm=e}) $*$
RestSMtoS@SMUSICIAN(**Current**,{pfm=p})

Note that a class whose only parent is STUDENT will likely need the first export clause in STUDENT to prove Implication of S_P. The second export clause is required in addition by SMUSICIAN, which demands a description of the state shared between S@STUDENT and M@MUSICIAN. The constructor's Body Verification further needs the export clause in PERSON:

{**Current**.age $\hookrightarrow$ _$*$**Current**.exams $\hookrightarrow$ _$*$**Current**.performances $\hookrightarrow$ _}
Precursor{STUDENT}(a,e)
{S@STUDENT(**Current**,{age=a;exm=e}) $*$ **Current**.performances $\hookrightarrow$ _}
{P@PERSON(**Current**,{age=a}) $*$ RestStoP@STUDENT(**Current**,{exm=e}) $*$
Current.performances $\hookrightarrow$ _}
{**Current**.age $\hookrightarrow$ a $*$ RestStoP@STUDENT(**Current**,{exm=e}) $*$
Current.performances $\hookrightarrow$ _}
Precursor{MUSICIAN}(a,p)
{M@MUSICIAN(**Current**,{age=a;pfm=p}) $*$
RestStoP@STUDENT(**Current**,{exm=e})}
{P@PERSON(**Current**,{age=a}) $*$ RestMtoP@MUSICIAN(**Current**,{pfm=p}) $*$
RestStoP@STUDENT(**Current**,{exm=e})}
{SM@SMUSICIAN(**Current**,{age=a;exm=e;pfm=p})}

Note that class SMUSICIAN would not have needed exported information if it simply overrode everything. The same is true for proof systems with less abstraction where method bodies are reverified in subclasses.

Although subclasses are constrained by axiom clauses, specification overhead can be reduced if axiom-based view shifts are used. For example, in its Body verification, *do_exam_performance* infers SM-specs for *take_exam* and *perform*:

{SM(**Current**,{age=a;exm=e;pfm=p})}
{S(**Current**,{age=a;exm=e}) $*$ RestSMtoS(**Current**,{pfm=p})}
Current.take_exam()
{S(**Current**,{age=a;exm=e+1}) $*$ RestSMtoS(**Current**,{pfm=p})}
{SM(**Current**,{age=a;exm=e+1;pfm=p})}
{M(**Current**,{age=a;pfm=p}) $*$ RestSMtoM(**Current**,{exm=e+1})}
Current.perform()
{M(**Current**,{age=a;pfm=p+1}) $*$ RestSMtoM(**Current**,{exm=e+1})}
{SM(**Current**,{age=a;exm=e+1;pfm=p+1})}

Such specifications are guaranteed to be implemented by all subclasses, so no dynamic type information is needed. Yet the system is still flexible – a subclass can always choose to satisfy such constraints vacuously by defining selected apf entries as false. Class DCell in [3] provides an example of this.

6 Limitations

Only shared multiple inheritance was considered. It appears to be significantly harder to accommodate replicated multiple inheritance, where (some) common ancestor state is not shared. Initial investigation shows that the fine-grained

```
class PERSON
define P@PERSON(x,{age=a}) as x.age ↪ a
export ∀x,a· P@PERSON(x,{age=a}) ⇔ x.age ↪ a where {}
feature
    introduce PERSON(a: int)
    dynamic {Current ↪ _}_{P(Current,{age=a})}
    do Current.age := a end

    introduce age(): int
    dynamic {P(Current,{age=a})}_{P(Current,{age=a}) ∗ Result = a}
    do Result := Current.age end

    introduce set_age(a: int)
    dynamic {P(Current,{age=_})}_{P(Current,{age=a})}
    do Current.age := a end

    introduce celebrate_birthday()
    dynstat {P(Current,{age=a})}_{P(Current,{age=a+1})}
    do tmp: int tmp := Current.age() tmp := tmp+1 Current.set_age(tmp) end

    age: int
end

class STUDENT inherit PERSON define
P@STUDENT(x,{age=a}) as P@PERSON(x,{age=a})
S@STUDENT(x,{age=a;exm=e}) as P@STUDENT(x,{age=a}) ∗ x.exams ↪ e
RestStoP@STUDENT(x,{exm=e}) as x.exams ↪ e
export ∀x,a,e· [P@STUDENT(x,{age=a}) ∗ RestStoP@STUDENT(x,{exm=e})] ⇔
                S@STUDENT(x,{age=a;exm=e}) where {}
       ∀x,a· P@STUDENT(x,{age=a}) ⇔ P@PERSON(x,{age=a}) where {}
axiom S_P: ∀a, e· S(Current,{age=a;exm=e}) ⇔
                    [P(Current,{age=a}) ∗ RestStoP(Current,{exm=e})]
feature
    introduce STUDENT(a: int, e: int)
    dynamic {Current.age ↪ _∗Current.exams ↪ _}_{S(Current,{age=a;exm=e})}
    do Precursor{PERSON}(a) Current.exams := e end

    introduce exams(): int dynamic
    {S(Current,{age=a;exm=e})}_{S(Current,{age=a;exm=e}) ∗ Result = e}
    do Result := Current.exams end

    introduce take_exam()
    dynamic {S(Current,{age=a;exm=e})}_{S(Current,{age=a;exm=e+1})}
    do tmp: int tmp := Current.exams Current.exams := tmp + 1 end

    inherit celebrate_birthday()
    dynstat {P(Current,{age=a})}_{P(Current,{age=a+1})}

    exams: int
end
```

Fig. 5. The PERSON and STUDENT classes.

class SMUSICIAN **inherit** STUDENT MUSICIAN
define
P@SMUSICIAN(x,{age=a}) **as** P@PERSON(x,{age=a})
S@SMUSICIAN(x,{age=a;exm=e}) **as** S@STUDENT(x,{age=a;exm=e})
M@SMUSICIAN(x,{age=a;pfm=p}) **as** M@MUSICIAN(x,{age=a;pfm=p})
SM@SMUSICIAN(x,{age=a;exm=e;pfm=p}) **as** P@PERSON(x,{age=a}) ∗
RestStoP@STUDENT(x,{exm=e}) ∗ RestMtoP@MUSICIAN(x,{pfm=p})
RestStoP@SMUSICIAN(x,{exm=e}) **as** RestStoP@STUDENT(x,{exm=e})
RestMtoP@SMUSICIAN(x,{pfm=p}) **as** RestMtoP@MUSICIAN(x,{pfm=p})
RestSMtoS@SMUSICIAN(x,{pfm=p}) **as** RestMtoP@MUSICIAN(x,{pfm=p})
RestSMtoM@SMUSICIAN(x,{exm=e}) **as** RestStoP@STUDENT(x,{exm=e})
axiom
SM_S: ∀a, e, p· SM(**Current**,{age=a;exm=e;pfm=p}) ⇔
[S(**Current**,{age=a;exm=e})∗RestSMtoS(**Current**,{pfm=p})]
SM_M: ∀a, e, p· SM(**Current**,{age=a;exm=e;pfm=p}) ⇔
[M(**Current**,{age=a;pfm=e})∗RestSMtoM(**Current**,{exm=e})]
feature
introduce SMUSICIAN(a: int, e: int, p: int)
dynamic {**Current**.age ↪ _∗**Current**.exams ↪ _∗**Current**.performances ↪ _}_
{SM(**Current**,{age=a;exm=e;pfm=p})}
do Precursor{STUDENT}(a,e) **Precursor**{MUSICIAN}(a,p) **end**

introduce do_exam_performance()
dynstat {SM(**Current**,{age=a;exm=e;pfm=p})}_
{SM(**Current**,{age=a;exm=e+1;pfm=p+1}
do Current.take_exam() **Current**.perform() **end**

inherit celebrate_birthday()
dynstat {P(**Current**,{age=a})}_{P(**Current**,{age=a+1})}
end

Fig. 6. The SMUSICIAN class.

replication offered by Eiffel and its 'select' clauses breaks behavioral subtyping in the presence of upcasts (e.g. in assignments and argument passing). In C++, replication is coarser-grained and happens on the subobject level. Upcasts must specify a subobject unambiguously, so their semantics should force view shifts in the logic. Replicated multiple inheritance has been criticized by some (e.g. [10]) and the authors are uncertain about its use in C++. It is rarely used in Eiffel, however, where non-conforming inheritance or composition can be used instead.

7 Conclusions and related work

The presented proof system offers a sound way to verify various forms and uses of shared multiple inheritance, which include the intertwining of ancestor functionality and diamond inheritance. By virtue of extending Parkinson and Bierman's

system, the examples in [3] illustrate that it can also deal with behavior extension, restriction and modification, as well as representation replacement in subclasses. It supports view shifting in the logic in two complementary ways, thereby permitting flexible abstraction, access control, savings in specification, and verification of code reuse. The system is modular and every method body is verified only once. No other proof system we are aware of provides these features.

This work is founded upon and therefore closely related to separation logic [8, 7], which offers local reasoning for heap-manipulating programs, and its application to object-orientation [9]. Separation logic and especially its frame rule support concise proofs at high levels of abstraction.

Our approach uses Parkinson and Bierman's apfs – a separation logic abstraction mechanism for reasoning about inheritance [9, 4, 3]. The system also builds upon other ideas for single-inheritance programs in [3]: the distinction between static and dynamic method specifications and the formalization of proof obligations resulting from this distinction. The examples in [3] furthermore demonstrate how our proof system can be applied.

The system for multiple inheritance in [12] also uses separation logic; it does not support the modular reasoning and abstraction facilitated by apfs. A different but compatible approach is taken in [13], where behavioral subtyping of methods is verified lazily, i.e. only to the extent demanded by client code.

Interface inheritance is easily handled by our proof system: an interface is simply an abstract class with only abstract methods and no fields. Many verification tools for object-oriented programs provide support for interface inheritance. jStar [6] includes such support and uses separation logic with abstract predicate families. We will soon extend it with export and axiom clauses for added expressiveness. The ability to constrain subclasses with axioms can even solve problems unrelated to view shifting in single inheritance programs[3]. We further plan to build upon jStar to mechanize our proof system for automated verification of Eiffel programs.

Spec# [14] and the JML [15] toolset also offer facilities to specify and verify interface inheritance. Both use pure expressions of the programming language for specification, and follow an object invariant-based approach to verification. Object invariants and axioms both constrain subclasses, but unlike object invariants, axioms do not specify consistency properties of objects which must be respected by public methods.

The separation logic proof system for single inheritance by Chin et al. [16] uses an object format which supports full and partial views of an object. These views are used for lossless upcasting and method specifications. Views in our work are generalized abstractions and not tightly coupled to classes or object formats.

Finally, our export clauses are somewhat related to the lemmas of the separation logic-based VeriFast tool [17]. The use of export information in our logic

[3] Matthew Parkinson pointed out in private communication that the informal discussion right before Section 5.1 and in Section 5.5 of [3] can be made rigorous by using our export and axiom clauses.

loosely corresponds to reasoning about a sequence of lemma invocations in ghost statements.

Acknowledgements
Special thanks to Matthew Parkinson, Peter O'Hearn, Bertrand Meyer, Sebastian Nanz, Carlo Furia and Martin Nordio for feedback on this work.

References

1. ECMA International: Standard ECMA-367. Eiffel: Analysis, Design and Programming Language. 2nd edn. (June 2006)
2. Cardelli, L.: A semantics of multiple inheritance. Inf. Comput. **76**(2-3) (1988) 138–164
3. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL '08, New York, NY, USA, ACM (2008) 75–86
4. Parkinson, M., Bierman, G.: Separation logic and abstraction. SIGPLAN Not. **40**(1) (2005) 247–258
5. Ellis, M.A., Stroustrup, B.: The annotated C++ reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)
6. Distefano, D., Parkinson J, M.J.: jStar: towards practical verification for Java. In: OOPSLA '08, New York, NY, USA, ACM (2008) 213–226
7. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. Logic in Computer Science, Symposium on **0** (2002) 55–74
8. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: CSL '01, London, UK, Springer-Verlag (2001) 1–19
9. Parkinson, M.J.: Local reasoning for Java. PhD thesis, University of Cambridge, Computer Laboratory. Technical Report UCAM-CL-TR-654 (November 2005)
10. Sakkinen, M.: Disciplined Inheritance. In: ECOOP. (1989) 39–56
11. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: ECOOP '08, Berlin, Heidelberg, Springer-Verlag (2008) 387–411
12. Luo, C., Qin, S.: Separation Logic for Multiple Inheritance. Electr. Notes Theor. Comput. Sci. **212** (2008) 27–40
13. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Incremental reasoning for multiple inheritance. In: IFM '09, Berlin, Heidelberg, Springer-Verlag (2009) 215–230
14. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview, Springer (2004) 49–69
15. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3) (2006) 1–38
16. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular OO verification with separation logic. SIGPLAN Not. **43**(1) (2008) 87–99
17. Jacobs, B., Piessens, F.: The VeriFast Program Verifier. Technical Report CW-520, Katholieke Universiteit Leuven (August 2008)

Appendix

Proof system semantics

An outline of the semantics and soundness proof follows. Most of our system's semantics is identical to that of Parkinson and Bierman's system in [3]. The most interesting difference is the treatment of export and axiom information in the soundness proof of the program verification rule (Theorem 12 below).

The semantics of the logical formula is defined in terms of a state σ, an interpretation of predicate symbols $\mathcal{I}$, and an interpretation of logical variables $\mathcal{L}$. The interpretation $\mathcal{I}$ maps predicate names to their definitions, whereas a definition maps a list of arguments to a set of states:

$$\mathcal{I} : \mathsf{Preds} \rightarrow (\mathsf{Vals}^* \rightarrow \mathcal{P}(\Sigma))$$
$$\mathcal{L} : \mathsf{Vars} \rightarrow \mathsf{Vals}$$

Predicates are defined in the standard way:

$$\sigma, \mathcal{I}, \mathcal{L} \models \mathrm{pred}(\overline{\mathrm{X}}) \Leftrightarrow \sigma \in (\mathcal{I}(\mathrm{pred})(\mathcal{L}(\overline{\mathrm{X}})))$$

Definition 5. $\mathcal{I} \models \Delta$ iff $\sigma, \mathcal{I}, \mathcal{L} \models \Delta$ for all σ and $\mathcal{L}$.

One can show that every set of disjoint apf and w-predicate definitions is satisfiable:

Lemma 6. For any set of definitions $\mathrm{W}_1, \ldots \mathrm{W}_\mathrm{m}$, $\mathrm{D}_1, \ldots \mathrm{D}_\mathrm{n}$ where W_i has form $\mathrm{w}_\mathrm{i}(\overline{\mathrm{x}_\mathrm{i}}) = \mathrm{Q}_\mathrm{i}$ and D_j is listed in class G_j, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models [\bigwedge_{\mathrm{i}\in 1..\mathrm{m}} \forall \overline{\mathrm{x}_\mathrm{i}} \cdot\ \mathrm{w}_\mathrm{i}(\overline{\mathrm{x}_\mathrm{i}}) \Leftrightarrow \mathrm{Q}_\mathrm{i}] \wedge [\bigwedge_{\mathrm{j}\in 1..\mathrm{n}} apf_{\mathrm{G}_\mathrm{j}}(\mathrm{D}_\mathrm{j})]$ provided that no two distinct definitions in the set define the same predicate.

The semantics of our proof system's judgements is defined next. We do not define the semantics of $\vdash_e$ and $\vdash_a$ explicitly, since we work with their premises (valid logical formulae whose existence is guaranteed) instead. For triples, the usual partial-correctness semantics for separation logic is used: if the precondition holds in the start state, then 1) the statements will not fault (access unallocated memory, for example), and 2) if the statements terminate, then the postcondition holds in the resulting state.

Definition 7. $\mathcal{I} \models_\mathrm{n} \{\mathrm{P}\}\bar{\mathrm{s}}\{\mathrm{Q}\}$ iff whenever $\sigma, \mathcal{I}, \mathcal{L} \models \mathrm{P}$ then $\forall \mathrm{m} \leq \mathrm{n} \cdot$

1. $\sigma,\bar{\mathrm{s}} \longrightarrow^\mathrm{m} \mathsf{fault}$ does not hold, and
2. if $\sigma,\bar{\mathrm{s}} \longrightarrow^\mathrm{m} \sigma',\epsilon$ then $\sigma', \mathcal{I}, \mathcal{L} \models \mathrm{Q}$

The index n deals with mutual recursion in method definitions. $\mathcal{I} \models_\mathrm{n} \Gamma$ means that all methods in Γ meet their specifications when executed for at least n steps.

Definition 8 (Method verification semantics)**.** If m in G is non-abstract, let $\bar{\mathrm{s}}$ denote its body.

$\mathcal{I}, \Gamma \models_0 \mathrm{G.m} \mapsto (\overline{\mathrm{x}},\{\mathrm{P}\}\_\{\mathrm{Q}\})$ always holds.

$\mathcal{I}, \Gamma \models_{\mathrm{n}+1} \mathrm{G.m} \mapsto (\overline{\mathrm{x}},\{\mathrm{P}\}\_\{\mathrm{Q}\})$ iff
$\mathcal{I} \models_\mathrm{n} \Gamma \Rightarrow \mathcal{I} \models_{\mathrm{n}+1} \{\mathrm{P} * \mathbf{Current} : \mathrm{G}\}\bar{\mathrm{s}}\{\mathrm{Q}\}$ if m is non-abstract in G and true otherwise.

$\mathcal{I}, \Gamma \models_0 \mathrm{G::m} \mapsto (\overline{\mathrm{x}},\{\mathrm{S}\}\_\{\mathrm{T}\})$ always holds.
$\mathcal{I}, \Gamma \models_{\mathrm{n}+1} \mathrm{G::m} \mapsto (\overline{\mathrm{x}},\{\mathrm{S}\}\_\{\mathrm{T}\})$ iff
$\quad\quad\quad\quad\quad\quad \mathcal{I} \models_{\mathrm{n}} \Gamma \Rightarrow \mathcal{I} \models_{\mathrm{n}+1} \{\mathrm{S}\}\overline{\mathrm{s}}\{\mathrm{T}\}$

$\mathcal{I} \models_{\mathrm{n}} \Gamma$ iff $\forall \mathrm{methodspec} \in \Gamma \cdot\ \mathcal{I}, \Gamma \models_{\mathrm{n}} \mathrm{methodspec}$

We next define the semantics of the statement judgement.

Definition 9. $\Delta; \Gamma \models \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$ iff for all $\mathcal{I}$ and n, if $\mathcal{I} \models \Delta$ and $\mathcal{I} \models_{\mathrm{n}} \Gamma$, then $\mathcal{I} \models_{\mathrm{n}+1} \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$

In other words, for all interpretations which satisfy the assumptions Δ, if all methods in Γ meet their specifications for at least n steps, then $\overline{\mathrm{s}}$ meets its specification for at least n + 1 steps.

The judgements are sound with respect to their semantics.

Lemma 10.

1. If $\Delta; \Gamma \vdash_m \ldots \mathrm{m} \ldots$, then $\forall \mathcal{I} \cdot$ if $\mathcal{I} \models \Delta$ then for all n and every spec of m we have $\mathcal{I}, \Gamma \models_{\mathrm{n}} \mathrm{spec}$
2. If $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$ then $\Delta; \Gamma \models \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$

Whenever a judgement is derivable under weak assumptions, it can also be derived under stronger ones.

Lemma 11.

1. If $\Delta; \Gamma \vdash_m \ldots \mathrm{m} \ldots$ and $\Delta' \Rightarrow \Delta$, then $\Delta'; \Gamma \vdash_m \ldots \mathrm{m} \ldots$
2. If $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$ and $\Delta' \Rightarrow \Delta$, then $\Delta'; \Gamma \vdash_s \{\mathrm{P}\}\overline{\mathrm{s}}\{\mathrm{Q}\}$
3. If $\Delta_{APF}, \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L}$ and $\Delta' \Rightarrow \Delta_{APF}$, then If $\Delta', \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L}$

Finally, here is the soundness statement and detailed proof sketch of the program verification rule.

Theorem 12. If a program and its main body $\overline{\mathrm{s}}$ can be proved with the program verification rule, then $\forall \mathcal{I}, \mathrm{n} \cdot\ \mathcal{I} \models_{\mathrm{n}} \{\mathrm{true}\}\overline{\mathrm{s}}\{\mathrm{true}\}$.
Proof.

1. *The goal.* We have to prove $\forall \mathcal{I}, \mathrm{n} \cdot\ \mathcal{I} \models_{\mathrm{n}} \{\mathrm{true}\}\overline{\mathrm{s}}\{\mathrm{true}\}$, which abbreviates $\forall \mathcal{I}, \mathrm{n} \cdot$ whenever σ, $\mathcal{I}$, $\mathcal{L} \models \mathrm{true}$, then $\forall \mathrm{m} \leq \mathrm{n} \cdot$ 1) $\sigma, \overline{\mathrm{s}} \longrightarrow^{\mathrm{m}} \mathsf{fault}$ does not hold, and 2) if $\sigma, \overline{\mathrm{s}} \longrightarrow^{\mathrm{m}} \sigma', \epsilon$ then σ', $\mathcal{I}$, $\mathcal{L} \models \mathrm{true}$. This can be simplified to $\forall \mathrm{n} \cdot\ \sigma, \overline{\mathrm{s}} \longrightarrow^{\mathrm{n}} \mathsf{fault}$ does not hold.
2. *Strengthened assumptions.* Let $\Delta_T \stackrel{\mathrm{def}}{=} \bigwedge_{\mathrm{i} \in 1..\mathrm{t}} \mathit{apf}(\mathrm{L_i})$, where $\mathrm{L_1} \ldots \mathrm{L_t}$ are all classes in the program. By Lemma 11, we can strengthen the assumptions under which all classes and the main body have been verified. For every class $\mathrm{L_i}$, we have Δ_T, Δ_E, Δ_A; $\Gamma \vdash_c \mathrm{L_i}$, and $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \vdash_s \{\mathrm{true}\}\overline{\mathrm{s}}\{\mathrm{true}\}$ also holds for the main body $\overline{\mathrm{s}}$.

3. *The interpretation* $\mathcal{I}'$. Since $\Delta_T \wedge \Delta_E \wedge \Delta_A; \Gamma \vdash_s \{\text{true}\}\bar{\text{s}}\{\text{true}\}$, Lemma 10 guarantees $\Delta_T \wedge \Delta_E \wedge \Delta_A; \Gamma \models \{\text{true}\}\bar{\text{s}}\{\text{true}\}$. This abbreviates $\forall \mathcal{I}$,n· if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_{\text{n}} \Gamma$, then $\mathcal{I} \models_{\text{n+1}} \{\text{true}\}\bar{\text{s}}\{\text{true}\}$, which can be simplified to $\forall \mathcal{I}$,n· if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_{\text{n}} \Gamma$, then $\forall \text{m} \leq \text{n}+1 \cdot \sigma,\bar{\text{s}} \longrightarrow^{\text{m}} \mathsf{fault}$ does not hold. Now if we can find an $\mathcal{I}'$ such that $\mathcal{I}' \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\forall \text{n} \cdot \mathcal{I}' \models_{\text{n}} \Gamma$, then we can instantiate $\mathcal{I}$ to $\mathcal{I}'$ in the formula and simplify to obtain $\forall \text{n} \cdot \sigma,\bar{\text{s}} \longrightarrow^{\text{n}} \mathsf{fault}$ does not hold. Therefore $\mathcal{I}'$ serves as a witness that $\bar{\text{s}}$ will never fault, which is exactly our goal.
   Let $\mathcal{I}'$ be the interpretation whose existence is guaranteed by Lemma 6 for all the where and define clauses in the program. Clearly $\mathcal{I}' \models \Delta_T$. We next prove $\mathcal{I}' \models \Delta_E$ and then $\mathcal{I}' \models \Delta_A$.
4. *Satisfiability of* Δ_E. Consider an arbitrary export clause E = P **where** $\{\text{w}_1(\overline{\text{x}_1}) = \text{Q}_1; \ldots; \text{w}_\text{n}(\overline{\text{x}_\text{n}}) = \text{Q}_\text{n}\}$ in class L. Since $apf(\text{L}) \vdash_e \text{E}$, we know $[apf(\text{L}) \wedge (\bigwedge_{\text{i} \in 1..\text{n}} \forall \overline{\text{x}_\text{i}} \cdot \text{w}_\text{i}(\overline{\text{x}_\text{i}}) \Leftrightarrow \text{Q}_\text{i})] \Rightarrow \text{P}$. The interpretation $\mathcal{I}'$ satisfies the antecedent, so we also have $\mathcal{I}' \models \text{P}$. Therefore $\mathcal{I}' \models \Delta_E$, and $\mathcal{I}' \models \Delta_T \wedge \Delta_E$.
5. *Satisfiability of* Δ_A. We prove this by induction. If class G has children $\text{H}_1 \ldots \text{H}_\text{k}$, let $level(\text{G}) \stackrel{\text{def}}{=} 1 + \max(0, level(\text{H}_1), \ldots, level(\text{H}_\text{k}))$. Furthermore, $P(\text{n}) \stackrel{\text{def}}{=} \forall$ G in the program such that $level(\text{G}) \leq \text{n}$ and for all axiom clauses a: P in the listing of G, $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\text{G}, \text{a: P})$.
   - Base case. Consider an arbitrary class G with $level(\text{G}) \leq 1$ and an axiom clause a: P appearing in it. G has no subclasses, and
     (a) If G is abstract, there are no objects with dynamic type G or a subtype thereof, thus $axiominfo(\text{G}, \text{a: P})$ holds vacuously and $\Delta_T \wedge \Delta_E$ implies it.
     (b) If G is non-abstract, then the only objects whose dynamic type is a subtype of G are instances of G. Since $(\Delta_T \wedge \Delta_E \wedge \textbf{Current} : \text{G}) \Rightarrow \text{P}$ by the Implication premise, we therefore also know $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\text{G}, \text{a: P})$.

     Thus $P(1)$ holds.
   - Step case. Suppose $P(\text{n})$ holds. Now consider a class G at level n+1 with axiom clause a: P. Every child H of G must list a, say a: Q. By the induction hypothesis we know $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\text{H}, \text{a: Q})$, and by the Parent Consistency premise of a: Q we know $(\Delta_T \wedge \Delta_E \wedge \text{Q}) \Rightarrow \text{P}$. Therefore $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\text{H}, \text{a: P})$. We have $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\text{G}, \text{a: P})$ if G is abstract, and the same holds if G is non-abstract since the Implication premise of a: P guarantees $(\Delta_T \wedge \Delta_E \wedge \textbf{Current} : \text{G}) \Rightarrow \text{P}$. Thus $P(\text{n+1})$ holds.

   So $\mathcal{I}' \models \Delta_T \wedge \Delta_E \wedge \Delta_A$.
6. *Wrapping up.* We still have to prove $\forall \text{n} \cdot \mathcal{I}' \models_{\text{n}} \Gamma$. Let m be an arbitrary method in the program. Since $\Delta_T \wedge \Delta_E \wedge \Delta_A; \Gamma \vdash_m \text{m}$, by Lemma 10 we know for all n and every spec of m that $\mathcal{I}', \Gamma \models_{\text{n}} \text{spec}$. Thus $\forall \text{n} \cdot \forall \text{methodspec} \in \Gamma \cdot \mathcal{I}', \Gamma \models_{\text{n}} \text{methodspec}$, in other words $\forall \text{n} \cdot \mathcal{I}' \models_{\text{n}} \Gamma$. □