Reasoning about Multiple Related Abstractions with MultiStar

Stephan van Staden
ETH Zurich, Switzerland
Stephan.vanStaden@inf.ethz.ch

Cristiano Calcagno
Imperial College, London and Monoidics Ltd
ccris@doc.ic.ac.uk

Abstract

Encapsulated abstractions are fundamental in object-oriented programming. A single class may employ multiple abstractions to achieve its purpose. Such abstractions are often related and combined in disciplined ways. This paper explores ways to express, verify and rely on logical relationships between abstractions. It introduces two general specification mechanisms: *export clauses* for relating abstractions in individual classes, and *axiom clauses* for relating abstractions in a class and all its descendants. MultiStar, an automatic verification tool based on separation logic and abstract predicate families, implements these mechanisms in a multiple inheritance setting. Several verified examples illustrate MultiStar's underlying logic. To demonstrate the flexibility of our approach, we also used MultiStar to verify the core iterator hierarchy of a popular data structure library.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects, Inheritance; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Separation logic, Multiple abstractions, Export clauses, Axiom clauses, Multiple inheritance

1. Introduction

The use of data abstractions is a hallmark of object-oriented (O-O) programming. A class is a typical example of such an abstraction. In interface or general multiple inheritance hierarchies, such as the one shown in Figure 1, a class can combine and maintain several abstractions offered by its parents. Although most examples of this paper involve abstractions in connection with inheritance, not all data abstractions

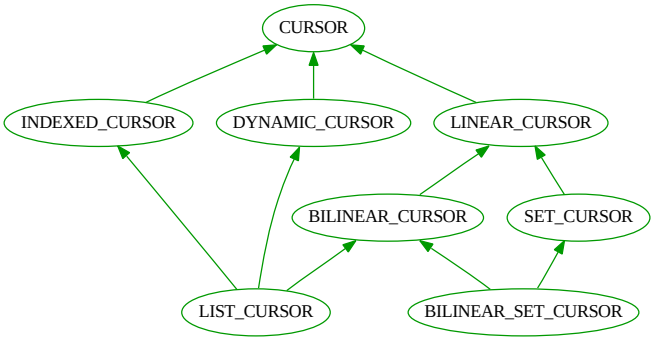

Figure 1. The core Gobo cursor hierarchy.

are directly coupled with language constructs. Classes use them for various purposes: to simplify how clients manipulate a class, to separate various concepts that are combined in a class, or to encourage or enforce particular call protocols. For example, a complex object with a long initialization phase can use the abstractions 'initializing' and 'ready', with methods applicable to an 'initializing' object, others to a 'ready' object, and some to both. Or algorithms can manipulate a mutable data structure under an 'immutable' abstraction, even if there is no interface making this explicit.

Relationships between data abstractions are important when reasoning about O-O code. This paper explores the problem of relating abstractions in an information hiding setting, where implementation details of abstractions are hidden from clients. Suppliers must therefore express and fulfill relationships between abstractions. If a class offers 'student' and 'person' abstractions (by using inheritance, or other means), for example, it might allow clients to convert a 'student' abstraction into a 'person' one. Clients can then manipulate the 'person' abstraction by calling e.g. library routines. After the manipulation, they might be allowed to convert back and assume that the number of exams the 'student' has taken is still the same. Specification mechanisms are needed to express and enforce the programmer's intentions about such relationships. This is especially important in multiple inheritance hierarchies where classes combine multiple abstractions in complicated ways.

A flexible mechanism for capturing data abstractions in O-O specifications is abstract predicate families, as introduced in [25, 27]. An *apf* (abstract predicate family) P pro-

vides a predicate name for an abstraction; each class C can define an *entry* predicate P_C. The definition of P_C describes how class C implements the apf P, and is hidden from other classes. For example, apfs S and P can be used to provide an abstraction of students and persons in a program respectively. Class STUDENT can define the entries $S_{STUDENT}$ and $P_{STUDENT}$, while other classes can define their apf entries differently. The predicate x.S(age: a, exm: e) describes object x under the 'student' abstraction: its age is 'a' and the number of exams taken is 'e'. If the dynamic type of x is STUDENT, then class STUDENT can use the fact that

$$\text{x.S(age: a, exm: e)} \Leftrightarrow \text{x.}S_{STUDENT}\text{(age: a, exm: e)}$$

In other words, the dynamic type of the first argument of an apf predicate (in this case the dynamic type of x) determines which apf entry applies. Apf predicates can therefore be seen to mirror dynamic dispatch of O-O programs in the logic. The apf mechanism is modular and exercises information hiding: only the class defining an apf entry knows the definition and can relate its entry to the apf predicate.

The relationship described before, namely that a 'student' abstraction can be converted into a 'person' one, can be expressed as follows:

$$\text{x.S(age: a, exm: e)} \Rightarrow \text{x.P(age: a)}$$

Allowing the back conversion without affecting the number of exams requires a stronger property that uses separation logic's $*$-connective:

$$\text{x.S(age: a, exm: e)} \Leftrightarrow [\text{x.P(age: a)} * \text{x.RestStoP(exm: e)}] \qquad \text{(A)}$$

where RestStoP abstracts the parts of a 'student' abstraction that are independent from and not included in a 'person' one. With this property, a client can now reason as the following proof outline shows:

```
{x.S(age: a, exm: e)}
{x.P(age: a) * x.RestStoP(exm: e)}
    {x.P(age: a)}
        // Manipulation of 'person' abstraction by library routines.
    {x.P(age: a+1)}
{x.P(age: a+1) * x.RestStoP(exm: e)}
{x.S(age: a+1, exm: e)}
```

The Frame rule of separation logic guarantees that the disjoint x.RestStoP(exm: e) remains unchanged. In essence, the client uses property (A) in combination with the Frame rule to infer an S-based specification for the library manipulation. It is not necessary to re-specify and re-verify the library – knowledge of the relationship saves specification overhead and keeps reasoning modular.

To which objects the property (A) applies is a design choice: a programmer might express that *selected* classes in a heterogeneous hierarchy fulfill the relationship, or that *all* classes in a homogeneous hierarchy fulfill it. We introduce two general specification mechanisms for the two cases: *export clauses* to express properties that hold for individual classes, and *axiom clauses* to describe properties of entire hierarchies. If class STUDENT specifies

export
∀x,a,e· x : STUDENT ⇒ [x.S(age: a, exm: e) ⇔
[x.P(age: a) ∗ x.RestStoP(exm: e)]]
where {}

then a client must know that an object's dynamic type is exactly STUDENT before using the information in reasoning. On the other hand, if class STUDENT specifies

axiom
S_P: ∀a,e· S(age: a, exm: e) ⇔ [P(age: a) ∗ RestStoP(exm: e)]

then clients can use the stronger implication

∀x,a,e· x <: STUDENT ⇒
[x.S(age: a, exm: e) ⇔ [x.P(age: a) ∗ x.RestStoP(exm: e)]]

Knowledge that the object's dynamic type is a subtype of STUDENT (including STUDENT) suffices to use the relationship. This is much more convenient for clients: a sound O-O type system will guarantee that if a variable has static type STUDENT and references an object, then the object's dynamic type will always be a subtype of STUDENT.

Axiom clauses offer a general facility to constrain the implementation of abstractions in subclasses. For example, class STUDENT can express that the number of exams a 'student' has taken is always non-negative, and that all subclasses should use its implementation of the 'student' abstraction:

axiom
exm_non_neg: ∀a,e· S(age: a, exm: e) ⇒ 0 ≤ e
S_constraint: ∀a,e· S(age: a, exm: e) ⇔ $S_{STUDENT}$(age: a, exm: e)

Axiom clause 'exm_non_neg' guarantees clients that $0 \leq$ e whenever they know x.S(age: a, exm: e) and x <: STUDENT. Subclass representation constraints such as the one expressed in the 'S_constraint' clause are useful for ensuring safe interaction between statically and dynamically dispatched calls on the same object – a pervasive pattern in O-O programs[1].

The claims made in export and axiom specifications must be checked to obtain sound reasoning. Whether or not a class fulfills axiom clauses often depends on properties of particular other classes, such as its parents. For this reason our proof system has a layered assumption structure: axiom verification can use export information of all classes in a program, and method verification can additionally use axiom information. Several examples in the paper show how export and axiom clauses are verified and applied in verification.

The paper contains a formalization of our proof system that extends the one of Parkinson and Bierman [27] with export/axiom clauses, abstract classes, abstract methods and shared multiple inheritance[2] where fields and methods of

[1] Matthew Parkinson pointed out in private communication that the informal discussion of representation constraints right before Section 5.1 and in Section 5.5 of [27] can be made rigorous by using export and axiom clauses.

[2] Inheritance with *virtual base classes* in C++ terminology [9].

common ancestor classes are not replicated in the descendant [8]. Apart from the use of apfs to support abstraction and information hiding, Parkinson and Bierman's system has the attractive property that it can verify a wide range of inheritance uses and abuses. Flexible handling of inheritance is vital in a proof system for multiple inheritance, since classes often interrelate methods and data from parents in complicated ways: methods can intertwine ancestor abstractions as the example in Section 2.1 shows, or abstractions can share parts in the case of diamond inheritance. Export and axiom clauses provide flexible ways to relate abstractions, thereby giving clients an integrated view of how methods affect abstractions from different inheritance paths. Very few proof systems exist for multiple inheritance, and no proof system we know of can facilitate reasoning about multiple related abstractions at the same level of abstraction as ours.

We implemented our proof system in MultiStar – a fully automatic verification tool. MultiStar has a two-tier architecture: a GUI front-end that translates Eiffel code and specifications into a simpler form for verification, and a language-independent back-end based on jStar [5] for reasoning. The front-end uses specifications written inside classes. It is easier to use than jStar, which currently does not have a front-end and requires separate code and specifications. Future front-ends for e.g. Java and C# can reuse the MultiStar back-end: with its support for interface inheritance, export/axiom clauses, abstract classes and abstract methods, a wide range of programs can be verified. As we shall see, the benefits of export and axiom clauses are not limited to multiple inheritance. Class DYNAMIC_CURSOR of Figure 1 uses only single inheritance, but cannot be verified with jStar because it relies on axiom information.

All the examples presented in this paper have been verified with MultiStar. To demonstrate the flexibility of our approach, we also used MultiStar to verify the Gobo data structure library's core iterator hierarchy of Figure 1. The complete code and specifications of the examples and Gobo case study are available online [10].

Outline Several examples illustrating the new specification mechanisms and proof system follow in Section 2. Section 3 presents the MultiStar tool, and Section 4 reports on the case study with Gobo iterator classes. A formal exposition of our proof system appears in Section 5. Section 6 concludes and mentions related work. The Appendix contains an overview of the formal semantics of our proof system, and a proof of soundness.

2. Examples

The examples are written in a language resembling Eiffel [8]. A class is divided into top-level sections. The **inherit** section lists its parent classes, the **define** section its apf entry definitions, the **export** and **axiom** sections its export and axiom clauses respectively, and methods and fields are written in the **feature** section. Empty sections are simply omitted. Two reserved program variables **Current** and **Result** denote the current object ('this') and the result of a function call respectively. **Current** is never **Void** ('null').

Methods have both **static** and **dynamic** specifications. A specification is written in pre-post form $\{P\}\_\{Q\}$, or alternatively $\{P_1\}\_\{Q_1\}$ **also** $\{P_2\}\_\{Q_2\}$ to indicate that both are satisfied (Section 5.4 shows how the **also** form abbreviates a single pre-post specification). A method's dynamic specification must be satisfied by all subclasses, and is used to verify dynamically-dispatched calls. A static specification describes properties about the particular method body, and is used to verify statically-dispatched calls, including **Precursor** ('super' or 'base') calls and direct calls x.C::m($\overline{e}$) in C++ style.

We omit the target of a method call or field assignment if it is **Current**. Similarly in the logic, $f \hookrightarrow e$ abbreviates **Current**.$f \hookrightarrow e$ (meaning that the **Current** object's field f has value e), and if the first argument of an apf predicate or entry is **Current** then it is simply omitted. We also employ the method specification shorthands of [27]: if only a static specification is listed, the dynamic specification is assumed to be exactly the same, and if only a dynamic specification is listed, then a static specification is derived by replacing each apf predicate whose first argument is **Current** with the entry predicate of the class. In other words, if the shorthand is used in class C, then $p(\overline{t : e})$ is replaced with $p_C(\overline{t : e})$. Specifications of non-constructor methods are furthermore propagated down the hierarchy: if a class does not explicitly list an inherited method, then it is assumed to have the same static and dynamic specifications as determined for the parent class. To avoid ambiguity, we require that if the method is available in multiple parents, then they must all have identical specifications for it.

The examples do not discuss details that are uninteresting from this paper's perspective, such as proofs of correctness of simple ancestor classes. Readers interested in details are referred to the formalization in Section 5; the paper of Parkinson and Bierman [27] also contains several examples.

2.1 Intertwining ancestor abstractions

Classes CELL and COUNTER are shown in Figure 2. CELL models mutable integer-valued cells and uses apf Cell, while COUNTER uses apf Cn. The apfs provide logical abstractions of mutable cells and counters respectively. Class CCELL in Figure 3, the focus of this example, inherits from CELL and COUNTER. It intertwines the functionality of its parents by overriding *set_value* to store the value and increment the count. It uses apf Cc to provide an abstraction of such objects in the logic, and 'grows' Cell_{CCELL} to accommodate method *set_value*, as we shall see.

The single export clause of CCELL relates the Cc, Cell and Cn abstractions. Only the predicate in front of **where** is exported for reasoning. Predicate definitions following **where** are used only to verify the clause and allow a class to hide implementation/representation details in its interface

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

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

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

    value: int
end

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

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

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

    count: int
end
```

Figure 2. The CELL and COUNTER classes.

without introducing new predicate families. For example, if we modify the representation of class CCELL, then the definition of Rest(x,v) can be changed without invalidating correctness proofs of client code.

To verify an export clause, we must prove that the exported predicate follows from the standard apf assumptions of the class and the predicate definitions after the **where** keyword. The proof for CCELL's export clause is trivial; for detail about standard apf assumptions, the reader is referred to Section 5.10. Note that export clauses are not verified for a particular dynamic type, since the standard apf assumptions of a class do not assume a particular one. It is therefore sound to use exported information to verify axiom clauses and methods of other classes, as we will do in later examples. However, nothing prevents the user from writing exported predicates as implications where the antecedent is of the form x : Type (see the export clause of CCELL). In this case information in the consequent can be applied only to objects satisfying this type constraint.

For the constructor we have to prove that its body satisfies the static specification (note that a **Precursor** call is syntactic sugar for a direct call):

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

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

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

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

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

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

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

Figure 3. The CCELL class and two library methods.

```
{value ↪ _ * count ↪ _}
    CELL::CELL(v)
{Cell_CELL(val: v) * count ↪ _}
    Precursor{COUNTER}()
{Cell_CELL(val: v) * Cn_COUNTER(cnt: 0)}
{Cc_CCELL(val: v, cnt: 0)}
```

The constructor body simply passes the needed fields to parent constructors and treats their internal representations abstractly thereafter.[3]

Method *value* is respecified in CCELL with Cell and Cc specifications. Since it inherits the body from CELL, we must prove that the new static specification is satisfied assuming the body's static specification of CELL. This method proof obligation is called Inheritance in the formalization, and readers that are unfamiliar with specification refinement are referred to Section 5.4 for detail. By applying the Frame rule (with $Cn_{COUNTER}$(cnt: c)) and then the rule of Consequence, we can derive each **also**-ed static specification,

[3] To simplify the formal presentation of proofs and make them more transparent, we mention fields explicitly in constructor preconditions. MultiStar injects them automatically – see Section 3.1 for more discussion.

which is sufficient to conclude the proof[4]. Another proof obligation for *value* is Behavioral subtyping, where we must show that the dynamic specification listed in CELL follows from the new one, i.e. that CCELL maintains the old specification. For the proof, we first 'choose' the Cell dynamic spec[5] and then remove the cnt tag by applying the Auxiliary Variable Elimination and Consequence rules. The application of Auxiliary Variable Elimination quantifies the variable c existentially in the pre- and postcondition, and the application of Consequence uses tag reduction information which is part of CCELL's standard apf assumptions.

Behavioral subtyping of *set_value* is similar. For its Body verification obligation, we must prove that both Cell_{CCELL} and Cc_{CCELL} static specifications are satisfied. The proof proceeds as follows:

{Cell_{CCELL}(val: _, cnt: c)}
    CCELL::increment()
{Cell_{CCELL}(val: _, cnt: c+1)}
{Cell_{CELL}(val: _) ∗ $\text{Cn}_{COUNTER}$(cnt: c+1)}
    CELL::set_value(v)
{Cell_{CELL}(val: v) ∗ $\text{Cn}_{COUNTER}$(cnt: c+1)}
{Cell_{CCELL}(val: v, cnt: c+1)}

An application of Consequence proves the other **also**-ed static spec and completes the proof[6]. As the body operates on state described by Cell_{CELL} and $\text{Cn}_{COUNTER}$, the proof obligations and separation logic's faulting semantics demand that we 'grow' Cell_{CCELL} to include both state parcels.

Now consider the two library routines at the bottom of Figure 3. The export clause contains the necessary information to prove the two triples:

{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: _)}

The proof of the second triple reduces and expands tags according to standard apf rules:

{true}
    cc := **new** CCELL(5)
{cc : CCELL ∗ cc.Cc(val: 5, cnt: 0)}
{cc : CCELL ∗ cc.Cell(val: 5, cnt: 0)}
{cc : CCELL ∗ cc.Cell(val: 5, cnt: _)}
{cc : CCELL ∗ cc.Cell(val: 5)}
    use_cell(cc,20)
{cc : CCELL ∗ cc.Cell(val: 20)}
{cc : CCELL ∗ cc.Cell(val: 20, cnt: _)}
{cc.Cc(val: 20, cnt: _)}

Information about cnt is lost in the postcondition, which is unavoidable because *use_cell* could call *set_value* more than once. 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. Also note that dynamic type information is required to use the exported relationships, since the subclasses of CCELL are not obliged to implement them.

[4] By Lemma 2 on page 13.

[5] By Lemma 1 on page 13.

[6] By Lemma 3 on page 13.

```
class CCEL2 inherit CELL COUNTER
define
x.Cell_CCEL2(val: v, cnt: c) as x.Cell_CELL(val: v) ∗ x.Cn_COUNTER(cnt: c)
x.Cn_CCEL2(cnt: c) as x.Cn_COUNTER(cnt: c)
export
    ∀x· x : CCEL2 ⇒ [∀v,c· x.Cell(val: v, cnt: c) ⇒ x.Cn(cnt: c)] where {}
feature
    introduce CCEL2(v: int)
    dynamic {value ↪ _ ∗ count ↪ _}_{Cell(val: v, cnt: 0)}
    do Precursor{CELL}(v); Precursor{COUNTER}() end

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

    override set_value(v: int)
    dynamic {Cell(val: _, cnt: c)}_{Cell(val: v, cnt: c+1)}
    do CCEL2::increment(); CELL::set_value(v) end

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

Figure 4. The CCEL2 class.

2.2 Access control and call protocols

Our 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. Its export clause relates the Cell and Cn abstractions in a one-directional way. 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 predicate. They must use exported information to get a Cn predicate, yet they lack information to change back after the call: no export or axiom clause is available to do this, and every method producing a Cell predicate requires one. The following proof attempt where cc2 : CCEL2 shows the problem:

{cc2.Cell(val: v, cnt: c)}
{cc2.Cn(cnt: c)}
    cc2.increment()
{cc2.Cn(cnt: c+1)}
{???}
{cc2.Cell(val: _, cnt: _)} // The weakest requirement of *set_value*.
    cc2.set_value(10)
{cc2.Cell(val: _, cnt: _)}

While the client has a Cell predicate, the argument tagged by cnt and returned by *count* reflects precisely how many times the value has been set. If the client tries to manipulate the count by calling *increment*, then it can never regain the needed capability to call *value* and *set_value*, and must forever treat the object as a simple counter in the code. The combination of abstract predicate relationships and method specifications enforces this protocol in verified code.

2.3 Diamond inheritance

Verification of multiple inheritance requires proper handling of data from several parent classes. Diamond inheritance

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

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

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

    introduce celebrate_birthday()
    static {P(age: a)}_{P(age: a+1)}
    do tmp: int; tmp := age(); tmp := tmp+1; set_age(tmp) end

    age: int
end

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

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

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

    exams: int
end
```

Figure 5. The PERSON and STUDENT classes.

complicates matters because common ancestor fields are shared. This is unproblematic for our proof system, although the abstraction of the shared data is typically lost. Diamond inheritance can moreover require relationships between several abstractions, which this example achieves with axiom clauses.

An axiom clause consists of a name and a predicate. The name identifies the clause and allows subclasses to refine the predicate. We propagate axiom clauses down the hierarchy to save specification overhead: if a class does not list an axiom clause with the same name as one in a parent, then it is assumed to list an identical clause. To avoid ambiguity in the presence of multiple inheritance, we require that if clauses with the same name are present in multiple parents, then they must all be identical. An axiom clause copied in this way

```
class SMUSICIAN inherit STUDENT MUSICIAN
define
x.P_SMUSICIAN(age: a) as x.P_PERSON(age: a)
x.S_SMUSICIAN(age: a, exm: e) as x.S_STUDENT(age: a, exm: e)
x.M_SMUSICIAN(age: a, pfm: p) as x.M_MUSICIAN(age: a, pfm: p)
x.SM_SMUSICIAN(age: a, exm: e, pfm: p) as x.P_PERSON(age: a) ∗
        x.RestStoP_STUDENT(exm: e) ∗ x.RestMtoP_MUSICIAN(pfm: p)
x.RestStoP_SMUSICIAN(exm: e) as x.RestStoP_STUDENT(exm: e)
x.RestMtoP_SMUSICIAN(pfm: p) as x.RestMtoP_MUSICIAN(pfm: p)
x.RestSMtoS_SMUSICIAN(pfm: p) as x.RestMtoP_MUSICIAN(pfm: p)
x.RestSMtoM_SMUSICIAN(exm: e) as x.RestStoP_STUDENT(exm: e)
axiom
    SM_S: ∀a, e, p· SM(age: a, exm: e, pfm: p) ⇔
                [S(age: a, exm: e) ∗ RestSMtoS(pfm: p)]
    SM_M: ∀a, e, p· SM(age: a, exm: e, pfm: p) ⇔
                [M(age: a, pfm: e) ∗ RestSMtoM(exm: e)]
feature
    introduce SMUSICIAN(a: int, e: int, p: int)
    dynamic {age ↪ _ ∗ exams ↪ _ ∗ performances ↪ _}_
            {SM(age: a, exm: e, pfm: p)}
    do Precursor{STUDENT}(a,e); Precursor{MUSICIAN}(a,p) end

    introduce do_exam_performance()
    static {SM(age: a, exm: e, pfm: p)}_{SM(age: a, exm: e+1, pfm: p+1)}
    do take_exam(); perform() end
end
```

Figure 6. The SMUSICIAN class.

is not refined in the subclass and automatically consistent with its parent versions. In the general case where a subclass refines an axiom clause, Parent consistency must be proven as indicated in the formalization.

The focus of this example is class SMUSICIAN, shown in Figure 6. It inherits from STUDENT and MUSICIAN, both which inherit from PERSON. The STUDENT and PERSON classes are shown in Figure 5; MUSICIAN is similar to STUDENT and not shown. A diamond is formed with PERSON at the top, and an instance of SMUSICIAN has one 'age' field, one *set_age* method, etc. under shared multiple inheritance semantics. The classes use axiom clauses to specify relationships between abstractions P, S, M and SM.

Since SMUSICIAN is non-abstract, we must prove that axiom SM_S holds for its direct instances. This proof obligation for axiom clauses is called Implication in the formalization. It holds indeed, since under the standard apf assumptions of SMUSICIAN, exported information of all classes, and the assumption **Current** : SMUSICIAN, we have:

```
SM(age: a, exm: e, pfm: p)
⇔   // Standard apf assumptions, Current : SMUSICIAN
SM_SMUSICIAN(age: a, exm: e, pfm: p)
⇔   // Standard apf assumptions.
P_PERSON(age: a) ∗ RestStoP_STUDENT(exm: e) ∗
  RestMtoP_MUSICIAN(pfm: p)
⇔   // Exported information from STUDENT.
S_STUDENT(age: a, exm: e) ∗ RestMtoP_MUSICIAN(pfm: p)
⇔   // Standard apf assumptions.
S_SMUSICIAN(age: a, exm: e) ∗ RestSMtoS_SMUSICIAN(pfm: p)
⇔   // Standard apf assumptions, Current : SMUSICIAN
S(age: a, exm: e) ∗ RestSMtoS(pfm: p)
```

The export clause in STUDENT is not closely connected to multiple inheritance. In fact, any class C inheriting from STUDENT which defines P_C to be P_{PERSON} and S_C to be $S_{STUDENT}$ will need the export clause to prove Implication of S_P, which we omit here for SMUSICIAN. Axiom verification frequently requires exported information of this kind. What is vital about the export clause in the shared multiple inheritance setting is that it isolates the shared ancestor state of SMUSICIAN, namely P_{PERSON}. This allows SMUSICIAN to relate ancestor abstractions in a fairly abstract way. Only the constructor's Body verification proof needs the export clause in PERSON[7]:

{age ↪ _ ∗ exams ↪ _ ∗ performances ↪ _}
Precursor{STUDENT}(a,e)
{$S_{STUDENT}$(age: a, exm: e) ∗ performances ↪ _}
{P_{PERSON}(age: a) ∗ RestStoP$_{STUDENT}$(exm: e) ∗ performances ↪ _}
{age ↪ a ∗ RestStoP$_{STUDENT}$(exm: e) ∗ performances ↪ _}
Precursor{MUSICIAN}(a,p)
{$M_{MUSICIAN}$(age: a, pfm: p) ∗ RestStoP$_{STUDENT}$(exm: e)}
{P_{PERSON}(age: a) ∗ RestMtoP$_{MUSICIAN}$(pfm: p) ∗
RestStoP$_{STUDENT}$(exm: e)}
{$SM_{SMUSICIAN}$(age: a, exm: e, pfm: p)}

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

Since **Current** in SMUSICIAN will always reference an object whose dynamic type is a subtype of SMUSICIAN, the Body verification proof of *do_exam_performance* can use axiom information to infer SM-specs for *take_exam* and *perform*:

{SM(age: a, exm: e, pfm: p)}
{S(age: a, exm: e) ∗ RestSMtoS(pfm: p)}
take_exam()
{S(age: a, exm: e+1) ∗ RestSMtoS(pfm: p)}
{SM(age: a, exm: e+1, pfm: p)}
{M(age: a, pfm: p) ∗ RestSMtoM(exm: e+1)}
perform()
{M(age: a, pfm: p+1) ∗ RestSMtoM(exm: e+1)}
{SM(age: a, exm: e+1, pfm: p+1)}

The specification overhead incurred by axiom clauses is offset by specification inference gains: SM, S and M specifications can be inferred for *age*, *set_age* and *celebrate_birthday*, while SM specifications can be inferred for *exams*, *take_exam*, *performances* and *get_performance* – a total of 13 specifications for methods of SMUSICIAN. These inferred specifications are guaranteed to be implemented by all subclasses, and no dynamic type information is needed to use them[8]. 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 [27] provides an example of this.

3. MultiStar

This section sketches notable aspects of the MultiStar implementation. MultiStar has a two-tier architecture: a front-end that translates Eiffel programs and specifications into a simpler form for verification, and a language-independent back-end based on jStar [5] which implements our proof system.

3.1 Front-end

The front-end provides a graphical user interface within the EVE integrated development environment, and is part of the standard EVE download [10]. It translates Eiffel code and specifications into the back-end's input format, and provides access to verification results. Verification is triggered by picking and dropping an annotated class on the MultiStar tool. Class annotations consist of apf entry definitions, export/axiom clauses and method specifications.

To simplify the proofs and formalization in this paper, constructor preconditions explicitly mention fields and break information hiding. The front-end translation of MultiStar injects them automatically. For example, a user would write the specification of CCELL's constructor as

dynamic {true}_{Cc(val: v, cnt: 0)}

instead of

dynamic {value ↪ _ ∗ count ↪ _}_{Cc(val: v, cnt: 0)}

In detail, the front-end facilitates this by:

1. Using jStar's **new** statement, whose specification is given by the triple $\{\mathrm{true}\}\mathrm{x} := \mathbf{new}\ \mathrm{C}\{\mathrm{x} : \mathrm{C}\}$. This allows us to omit the fields in the dynamic precondition[9].
2. Adding all fields (including ancestor ones) to the static precondition when checking Body verification, and consuming all fields of a parent class and its ancestors right before the parent constructor is called. This is communicated to the back-end by emitting special instructions, and allows us to omit the fields in the static precondition.

The Dynamic dispatch proof obligation, which checks that the static and dynamic specifications are consistent with each other, is unaffected because fields are omitted in both

[7] Unless a class manipulates fields of its ancestors directly, this export clause would not be needed in languages where constructors of common ancestor classes cannot be called more than once.

[8] The technique used by Chin et al. [4] of inheriting static method specifications and deriving dynamic specifications from them implements "internal specification inference", i.e. a class infers and publishes dynamic specifications for its methods which external clients can use. In contrast to this, the technique of equipping classes with export/axiom clauses implements "external specification inference", i.e. clients infer specifications for methods based on published export/axiom information. A benefit of the external approach is that clients can infer valid specifications for library code without re-verifying it. For example, knowing only the inferred specifications for methods of class SMUSICIAN is not enough for proving the triple {x.SM(age: a, exm: e, pfm: p)}use_student(x){x.SM(age: a+1, exm: e+4, pfm: p)} without looking at the implementation of the library routine use_student(st: STUDENT) whose dynamic specification is {st.S(age: a, exm: e)}_{st.S(age: a+1, exm: e+4)}.

[9] The dynamic specification of the constructor is used for object initialization. x := **new** C($\overline{\mathrm{e}}$) abbreviates x := **new** C; x.C($\overline{\mathrm{e}}$) if x is not free in $\overline{\mathrm{e}}$.

static and dynamic preconditions. In languages where no fields are shared by ancestors, or constructors of common ancestors are called only once, the manipulation does not have to add ancestor fields to the static precondition and consume fields when a parent constructor is called. This is the approach jStar uses for Java verification. A front-end for C++ can use a similar technique because every ancestor constructor is called exactly once when virtual base classes are used [9].

3.2 Back-end

The MultiStar back-end extends jStar with support for export and axiom clauses, abstract classes and multiple inheritance. The latter two demand generalized method proof obligation checking.

3.2.1 Export and axiom clauses

The background theory used by the jStar theorem prover is encoded as a list of sequent rules. A sequent is of the form $\mathrm{P} \mid \mathrm{Q} \vdash \mathrm{R}$, meaning $(\mathrm{P} * \mathrm{Q}) \Rightarrow (\mathrm{P} * \mathrm{R})$. Each sequent rule has the form

$\mathrm{A} \mid \mathrm{B} \vdash \mathrm{C}$
if
$\mathrm{D} \mid \mathrm{E} \vdash \mathrm{F}$

If the prover is trying to prove a sequent that matches the rule's conclusion $\mathrm{A} \mid \mathrm{B} \vdash \mathrm{C}$, it suffices to prove the sequent where the rule's premise $\mathrm{D} \mid \mathrm{E} \vdash \mathrm{F}$ replaces the matched predicates. A new proof goal is thus obtained, and the proof is complete when the goal is of the form $\mathrm{G} \mid \mathrm{H} \vdash$. For details the reader is referred to [5].

Exported information is written as sets of implications. Before verifying an export clause, the background theory is temporarily extended with the definitions of all predicates in its **where** part. For each definition of the form w(x) = P, the following two rules are generated:

$\mid \mathrm{w}(\mathrm{x}) \vdash$ **if** $\mid \mathrm{P} \vdash$ and $\mid \vdash \mathrm{w}(\mathrm{x})$ **if** $\mid \vdash \mathrm{P}$

After all exported implications in the clause have been checked, the definitions are removed from the background theory. After all export clauses have been verified, each exported implication $\mathrm{P} \Rightarrow (\mathrm{Q}_1 * \ldots * \mathrm{Q}_\mathrm{n})$ is added to the background theory as a set of n rules, where rule $\mathrm{i} \in 1..\mathrm{n}$ has the form

$\mid \mathrm{P} \vdash \mathrm{Q}_\mathrm{i}$
if
$\mathrm{Q}_\mathrm{i} \mid \mathrm{Q}_1 * \ldots * \mathrm{Q}_{\mathrm{i}-1} * \mathrm{Q}_{\mathrm{i}+1} * \ldots * \mathrm{Q}_\mathrm{n} \vdash$

This rule form retains information about Q_i in its premise, and removal of Q_i from the goal sequent's right-hand side brings the proof closer to completion.

The background theory augmented with export information is then used to verify axiom clauses. The predicates in axiom clauses are written as implications. After all axiom clauses have been verified, an axiom implication $\mathrm{P} \Rightarrow (\mathrm{Q}_1 * \ldots * \mathrm{Q}_\mathrm{n})$ written in class C is encoded as n rules, with rule $\mathrm{i} \in 1..\mathrm{n}$ of the form

$\mid \mathrm{P} \vdash \mathrm{Q}_\mathrm{i}$
if
$\mathrm{Q}_\mathrm{i} \mid \mathrm{Q}_1 * \ldots * \mathrm{Q}_{\mathrm{i}-1} * \mathrm{Q}_{\mathrm{i}+1} * \ldots * \mathrm{Q}_\mathrm{n} \vdash \mathrm{x} <: \mathrm{C}$

where x is the pattern variable substituted for **Current**.

The background theory augmented with export and axiom information is then used for method verification.

3.2.2 Method proof obligations

The back-end accommodates abstract classes and abstract methods in addition to shared multiple inheritance. An abstract method has no body and hence no static specification. The back-end takes this into account when expanding specification shorthands. After shorthand expansion, verification of method m in class C proceeds as follows (the formalization contains details about the proof obligations):

- If m has a static specification and C can be instantiated (i.e. is non-abstract), then check Dynamic dispatch.
- If m has a body in C, then check Body verification.
- Always check Behavioral subtyping. This succeeds trivially if m is introduced in C: the set of dynamic specifications for m in C's parents is empty, and therefore all its elements are preserved by the new specification.
- If m has a static specification but no body in C, then check Inheritance.

The treatment subsumes interface inheritance – interfaces are treated as abstract classes with only abstract methods and no fields.

4. Case study

The Gobo data structure library [11] is an open-source Eiffel library covering data structures and algorithms. It contains classic data structures such as lists, stacks and sets, and provides several implementations of each structure. The library is stable and a popular choice among Eiffel developers.

Data structures such as lists and sets can be traversed with iterators. The iterator (or *cursor*) hierarchy is characterized by relatively simple algorithms and extensive use of multiple inheritance, which makes it an ideal candidate for evaluating the novel aspects of our proof system and its implementation. The core classes are shown in Figure 1: a LINEAR_CURSOR can traverse a data structure forwards, a BILINEAR_CURSOR can traverse both forwards and backwards, an INDEXED_CURSOR offers random data structure access with an integer position or index, and a DYNAMIC_CURSOR can modify the data structure being traversed.

We successfully verified the core cursor hierarchy of Figure 1 with MultiStar. The overall effort for specification and verification was five person-days. Most of the time was spent on finding and revising specifications, since we did not modify the code. Table 1 shows the experimental results. The to-

Class	LOC_1	LOC_2	Time(s)
BILINEAR_CURSOR	99	124	1.306
BILINEAR_SET_CURSOR	44	50	0.841
CURSOR	130	158	1.039
DYNAMIC_CURSOR	50	66	1.070
INDEXED_CURSOR	46	57	0.698
LINEAR_CURSOR	98	123	1.327
LIST_CURSOR	238	271	1.643
SET_CURSOR	38	44	0.738
8 classes	743	893	8.662

Table 1. Experimental results of the Gobo iterator case study. LOC_1 and LOC_2 denote the lines of code before and after specification respectively. MultiStar was executed on a 2.53 GHz Intel Core 2 Duo with 4 GB RAM.

```
abstract class DYNAMIC_CURSOR [G] inherit CURSOR [G]
feature
    introduce abstract replace(v: G) dynamic
    {Cursor(ds: d) ∗ d.DS(content: c1, iters: i) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c1)}_
    {Cursor(ds: d) ∗ d.DS(content: c2, iters: i) ∗
d.Replaced(ref: Current, value: v, newcontent: c2, oldcontent: c1, iters: i)}

    inherit item(): G static
    {Cursor(ds: d) ∗ d.DS(content: c, iters: i) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c)}_
    {Cursor(ds: d) ∗ d.DS(content: c, iters: i) ∗
      d.ItemAt(res: Result, ref: Current, iters: i, content: c)}

    introduce swap(other: DYNAMIC_CURSOR [G]) static
    {Cursor(ds: d) ∗ d.DS(content: c1, iters: i) ∗ other.Cursor(ds: d) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c1) ∗
      d.IsOff(res: False, ref: other, iters: i, content: c1)}_
    {Cursor(ds: d) ∗ d.DS(content: c2, iters: i) ∗ other.Cursor(ds: d) ∗
d.Swapped(ref1: Current, ref2: other, iters: i, oldcontent: c1, newcontent: c2)}
    do
        v: G; w: G;
        v := item(); w := other.item();
        replace(w); replace(v)
    end
end
```

Figure 7. A simplified extract of DYNAMIC_CURSOR

tal time taken by MultiStar is reported, which includes translating Eiffel code, expanding specification shorthands and checking all proof obligations.

Since iterators rely on properties of the data structures (containers) they traverse, we annotated the container classes with the required specifications. Particularly interesting are the axiom clauses that iterators demand. Consider for example the simplified extract of DYNAMIC_CURSOR in Figure 7. The [G] denotes that DYNAMIC_CURSOR has a generic parameter G. Method *swap* takes another cursor referencing the same container, and additionally requires that there are data elements (items) at both cursor positions (the cursors are not 'off'). The Body verification proof of *swap* uses several properties of containers that can be expressed as axioms, including the following one:

```
∀ r1,r2,iter1,iter2,i,c1,c2,c3 ·
[ItemAt(res: r1, ref: iter1, iters: i, content: c1) ∗
 ItemAt(res: r2, ref: iter2, iters: i, content: c1) ∗
 Replaced(ref: iter1, value: r2, newcontent: c2, oldcontent: c1, iters: i) ∗
 Replaced(ref: iter2, value: r1, newcontent: c3, oldcontent: c2, iters: i)]
⇒
Swapped(ref1: iter1, ref2: iter2, iters: i, oldcontent: c1, newcontent: c3)
```

This invariant property relates the ItemAt, Replaced and Swapped abstractions. Informally, it states that if a data structure has items r1 and r2 at iterators iter1 and iter2, and we replace the item at iter1 with r2 and the item at iter2 with r1, then the resulting data structure has the same contents as the original one except that the items at iter1 and iter2 have been swapped. The example involves neither multiple inheritance nor splitting of superclass and subclass state, and illustrates the generality and expressive power of axiom clauses.

The complete specifications and code of the case study are included in the download [10].

5. Formalization

This section contains a formal treatment of the programming language with specifications and its proof system. The language features abstract classes and multiple inheritance. Export and axiom specifications are supported. The proof system is based on the one of Parkinson and Bierman in [27]. For space reasons we focus mostly on the new extensions.

5.1 Language syntax

The grammar of our kernel language with multiple inheritance and specifications is shown in Figure 8. A sequence of c's is denoted by $\overline{c}$. The letters G and H are used for class names, p for apf names, t for tag names, w for auxiliary predicate names, a for axiom names, m for method names, and f for field names. Variables are denoted by u, x, y and z.

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.

A constructor in our formalization 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 (in MultiStar and the examples, specification shorthands are employed to achieve this). The shared semantics of multiple inheritance is used, which is popular in Eiffel [8] and known as inheritance with *virtual base classes* in C++ [9]. Common an-

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 ::= x.$\text{p}_G(\overline{\text{t}: \text{y}})$ **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($\overline{\text{e}}$) | y.m($\overline{\text{e}}$)	*Dynamically dispatched call*
| x := y.G::m($\overline{\text{e}}$) | y.G::m($\overline{\text{e}}$)	*Direct method call*
| x := **new** G	*Object allocation*
e ::= x | e + e | e = e | **Void** | 0 | 1 | 2 | ...	*Expression*
Type ::= int | bool | G	
Args ::= $\overline{\text{x}: \text{Type}}$	*Formal arguments*
Rt ::= ϵ | : Type	*Return type*

Figure 8. The kernel language grammar.

cestor fields are shared, and method overriding overrides all ancestor versions. To avoid ambiguity, a class can inherit a method only if its body (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.

5.2 Operational semantics

The shared semantics of multiple inheritance ensures that 1) only dynamic type information is needed at runtime (in contrast to what 'select' clauses of Eiffel's replicated inheritance demand), and 2) the usual semantics of casts can be adopted (in contrast to replicated inheritance in C++, where casting can change pointer values [9]).

The operational semantics is therefore similar to e.g. Java's 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.

5.3 Logic syntax and semantics

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

P, Q, S, T, Δ ::= $\forall$x·P | P$\Rightarrow$Q | false | e = e′ | x : G | x <: G | x.f $\hookrightarrow$ e | P $*$ Q

| x.p($\overline{\text{t}: \text{e}}$)	*Apf predicate*
| x.$\text{p}_G(\overline{\text{t}: \text{e}})$	*Apf entry*
| w($\overline{\text{x}}$)	*Auxiliary predicate*

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 first argument of an apf predicate or entry is written as a prefix. For apf predicates it is never **Void** because of the standard apf assumptions needed to produce an apf predicate from an entry (these are detailed in Section 5.10 below). Other arguments are tagged with names and form a set (i.e. they are order-independent), which is especially useful in the multiple inheritance setting. The reader is referred to [27] for an in-depth treatment of apfs.

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 [24, 26, 28] 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 formalization, the symbols P, Q, S and T are used for assertions and predicates, and Δ for assumptions.

5.4 Specification refinement

We expand on Parkinson and Bierman's formalization of specification refinement [27]. If the specification $\{\text{P}_1\}\_\{\text{Q}_1\}$ is refined by $\{\text{P}_2\}\_\{\text{Q}_2\}$, then any $\overline{\text{s}}$ that satisfies $\{\text{P}_1\}\_\{\text{Q}_1\}$ also satisfies $\{\text{P}_2\}\_\{\text{Q}_2\}$. Under the assumptions Δ, the specification $\{\text{P}_1\}\_\{\text{Q}_1\}$ is refined by $\{\text{P}_2\}\_\{\text{Q}_2\}$ if we can prove $\Delta \vdash \{\text{P}_1\}\_\{\text{Q}_1\} \implies \{\text{P}_2\}\_\{\text{Q}_2\}$, i.e. provide a proof tree with leaves $\Delta \vdash \{\text{P}_1\}\_\{\text{Q}_1\}$ and root $\Delta \vdash \{\text{P}_2\}\_\{\text{Q}_2\}$ built with the structural rules of separation logic (Consequence, Frame, Auxiliary Variable Elimination, Disjunction, and others). In the context of method specification refinement, Δ contains the standard apf assumptions of a class as well as export and axiom information of all other classes, and the Consequence and Frame rules are given by:

$$\frac{\Delta \Rightarrow (\text{P}' \Rightarrow \text{P}) \quad \Delta \vdash \{\text{P}\}\_\{\text{Q}\} \quad \Delta \Rightarrow (\text{Q} \Rightarrow \text{Q}')}{\Delta \vdash \{\text{P}'\}\_\{\text{Q}'\}} \text{ Consequence}$$

$$\frac{\Delta \vdash \{\text{P}\}\_\{\text{Q}\}}{\Delta \vdash \{\text{P} * \text{T}\}\_\{\text{Q} * \text{T}\}} \text{ Frame}$$

The Frame rule is applicable whenever **Result** $\notin$ *FV*(T), and expresses that disjoint portions of the heap stay unchanged.

Method specifications can be combined with **also** (Definition 1 in [27]):

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

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 $\textbf{also}_{i\in I}\ \{P_i\}\_\{Q_i\}$ denotes the specification $\{P_{e_1}\}\_\{Q_{e_1}\}$ **also** ... **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:

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

It always holds that $\Delta \vdash \{P\}\_\{Q\} \Longrightarrow \{\text{false}\}\_\{\text{true}\}$. Other useful lemmas involving **also** are given in Section 5.11. Finally, we use the abbreviation

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

in the formalization of the Dynamic dispatch proof obligation for methods in Section 5.9.

5.5 The specification environment

Most of the proof rules that 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 $G.a \mapsto P \in \Gamma$, $\Gamma(G.m) = (\overline{x},\{P\}\_\{Q\})$ for $G.m \mapsto (\overline{x},\{P\}\_\{Q\}) \in \Gamma$, and $\Gamma(G::m) = (\overline{x},\{S\}\_\{T\})$ for $G::m \mapsto (\overline{x},\{S\}\_\{T\}) \in \Gamma$.

5.6 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 relationships between apfs or their entries, and to expose apf entry definitions. Information can 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. This is captured by the following proof rule:

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

Since the assumptions about a class do not include assumptions about the exact dynamic type of **Current**, export information can be used to verify axioms and methods of other classes in a program.

5.7 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 methods. To simplify the treatment, we require that a class explicitly lists all axiom clauses applicable to it (in MultiStar and the examples, specification shorthands achieve this).

In the rule 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 (the Parent consistency [P.c.] obligation). A non-abstract class must also show that the predicate holds for its direct instances (the Implication [Imp.] obligation).

$$\frac{\begin{array}{ll}\forall i \in I \cdot \Gamma(H_i.a) = Q_i \ \wedge\ \forall j \in (1..n \setminus I) \cdot H_j.a \notin \mathit{dom}(\Gamma) & \\ (\Delta \wedge P) \Rightarrow \bigwedge_{i\in I} Q_i & \text{[P.c.]} \\ Ab{\neq}\epsilon \ \vee\ (\Delta \wedge \textbf{Current} : G) \Rightarrow P & \text{[Imp.]}\end{array}}{\Delta; \Gamma \vdash_a a: P \text{ in Ab G parents } H_1 \ldots H_n}$$

5.8 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. [26, 27]). 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 free in $\overline{e}$, the rules for result-returning calls are:

$$\frac{\Gamma(G.m) = (\overline{u},\{P\}\_\{Q\})}{\Delta; \Gamma \vdash_s \{P[y, \overline{e}/\textbf{Current}, \overline{u}] * y <: G\}\ x := y.m(\overline{e})\ \{Q[y, \overline{e}, x/\textbf{Current}, \overline{u}, \textbf{Result}]\}}$$

$$\frac{\Gamma(G::m) = (\overline{u},\{S\}\_\{T\})}{\Delta; \Gamma \vdash_s \{S[y, \overline{e}/\textbf{Current}, \overline{u}] * y \neq \textbf{Void}\}\ x := y.G::m(\overline{e})\ \{T[y, \overline{e}, x/\textbf{Current}, \overline{u}, \textbf{Result}]\}}$$

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*(T):

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

The rule of Consequence allows the use of assumptions Δ:

$$\frac{\Delta \Rightarrow (P' \Rightarrow P) \quad \Delta; \Gamma \vdash_s \{P\}\overline{s}\{Q\} \quad \Delta \Rightarrow (Q \Rightarrow Q')}{\Delta; \Gamma \vdash_s \{P'\}\overline{s}\{Q'\}}\ \text{Consequence}$$

5.9 Method verification

The rules for method verification in [27] 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 shown below requires no modification for multiple inheritance. A newly introduced method's static and dynamic specifications must be consistent[10] if the class is non-abstract, and its body must satisfy the static specification. These two requirements are captured by the Dynamic dispatch [D.d.] and Body verification [B.v.] proof obligations respectively.

$$\frac{\begin{array}{l}\mathrm{B} = \textbf{do}\ \overline{\mathrm{s}}\ \textbf{end}\\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\}\\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\}\\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\mathbf{Current}\,:\,\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} & \text{[D.d.]}\\ \Delta; \Gamma \vdash_s \{\mathrm{S_G}\}\overline{\mathrm{s}}\{\mathrm{T_G}\} & \text{[B.v.]}\end{array}}{\Delta; \Gamma \vdash_m \textbf{introduce}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ Ab\ G\ parents}\ \overline{\mathrm{H}}}$$

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

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

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; this is embodied in the Behavioral subtyping [B.s.] proof obligation. The other proof obligations are identical to those for method introduction above. The $\mathrm{H_1} \ldots \mathrm{H_n}$ are the immediate superclasses of G.

$$\frac{\begin{array}{l}\forall \mathrm{i} \in \mathrm{I}\cdot \Gamma(\mathrm{H_i.m}) = (\overline{\mathrm{x}},\{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\})\\ \forall \mathrm{j} \in (1..\mathrm{n} \setminus \mathrm{I})\cdot \mathrm{H_j.m} \notin \mathit{dom}(\Gamma)\\ \mathrm{B} = \textbf{do}\ \overline{\mathrm{s}}\ \textbf{end}\\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\}\\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\}\\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) & \text{[B.s.]}\\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\mathbf{Current}\,:\,\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} & \text{[D.d.]}\\ \Delta; \Gamma \vdash_s \{\mathrm{S_G}\}\overline{\mathrm{s}}\{\mathrm{T_G}\} & \text{[B.v.]}\end{array}}{\Delta; \Gamma \vdash_m \textbf{override}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ Ab\ G\ parents}\ \mathrm{H_1} \ldots \mathrm{H_n}}$$

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

[10] We establish that the dynamic specification follows from the static one when the dynamic type of **Current** is G. If the dynamic type of x is G, then the body of m in G will be executed if x.m is called. The static and dynamic specifications must be consistent with each other in this case, since the dynamic specification is used for reasoning about the call statement, whereas the body was verified only w.r.t. the static specification.

$$\frac{\begin{array}{l}\forall \mathrm{i} \in \mathrm{I}\cdot \Gamma(\mathrm{H_i.m}) = (\overline{\mathrm{x}},\{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\})\\ \forall \mathrm{k} \in (1..\mathrm{n} \setminus \mathrm{I})\cdot \mathrm{H_k.m} \notin \mathit{dom}(\Gamma)\\ \forall \mathrm{j} \in \mathrm{J}\cdot \Gamma(\mathrm{H_j::m}) = (\overline{\mathrm{x}},\{\mathrm{S_{H_j}}\}\text{-}\{\mathrm{T_{H_j}}\})\\ \forall \mathrm{l} \in (1..\mathrm{n} \setminus \mathrm{J})\cdot \mathrm{H_l::m} \notin \mathit{dom}(\Gamma)\\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\}\\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\}\\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) & \text{[B.s.]}\\ \Delta \vdash (\textbf{also}_{\mathrm{j}\in\mathrm{J}}\ \{\mathrm{S_{H_j}}\}\text{-}\{\mathrm{T_{H_j}}\}) \Longrightarrow \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} & \text{[Inh.]}\\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\mathbf{Current}\,:\,\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} & \text{[D.d.]}\end{array}}{\Delta; \Gamma \vdash_m \textbf{inherit}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ in\ Ab\ G\ parents}\ \mathrm{H_1} \ldots \mathrm{H_n}}$$

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.

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

5.10 Class and program verification

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.

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

Finally, here is the rule for program verification:

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

$\mathit{exportinfo}(\mathrm{P}\ \textbf{where} \ldots) \overset{\mathrm{def}}{=} \mathrm{P}$

$\mathit{axiominfo}(\mathrm{G, a: P}) \overset{\mathrm{def}}{=} \forall \mathrm{x} <: \mathrm{G} \cdot \mathrm{P[x/}\textbf{Current}\mathrm{]}$, 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 – its standard apf assumptions. It is adapted from [27] for tagged arguments:

$apf(\text{Ab } \textbf{class } \text{G} \ldots \textbf{define } \text{D}_1\ \text{D}_2 \ldots \text{D}_\text{n} \textbf{ export} \ldots \textbf{end}) \stackrel{\text{def}}{=}$
$apf_\text{G}(\text{D}_1) \wedge \ldots \wedge apf_\text{G}(\text{D}_\text{n})$

$apf_\text{G}(\text{x}.\text{p}_G(\text{Y}) \textbf{ as } \text{P}) \stackrel{\text{def}}{=}$
$FtoE(\text{p,G,Y}) \wedge EtoD(\text{x}.\text{p}_G(\text{Y}) \textbf{ as } \text{P}) \wedge (\forall \text{x} <: \text{G} \cdot TR(\text{p,x,Y}))$

$FtoE(\text{p,G,}\overline{\text{t}:\text{y}}) \stackrel{\text{def}}{=}$
$\forall \text{x,}\overline{\text{y}} \cdot \text{x} : \text{G} \Rightarrow [\text{x}.\text{p}(\overline{\text{t}:\text{y}}) \Leftrightarrow \text{x}.\text{p}_G(\overline{\text{t}:\text{y}})]$

$EtoD(\text{x}.\text{p}_G(\overline{\text{t}:\text{y}}) \textbf{ as } \text{P}) \stackrel{\text{def}}{=}$
$\forall \text{x,}\overline{\text{y}} \cdot \text{x}.\text{p}_G(\overline{\text{t}:\text{y}}) \Leftrightarrow \text{P}$

$TR(\text{p,x,}\overline{\text{t}:\text{y}}) \stackrel{\text{def}}{=}$
$\bigwedge_{\overline{\text{t}':\text{y}'} + \overline{\text{t}'':\text{y}''} \equiv \overline{\text{t}:\text{y}}} \forall \overline{\text{y}'} \cdot \text{x}.\text{p}(\overline{\text{t}':\text{y}'}) \Leftrightarrow \text{x}.\text{p}(\overline{\text{t}':\text{y}'} + \overline{\text{t}'':\_})$

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

Theorem. The program verification rule is sound. (The proof, sketched in the Appendix, depends on the layered assumption structure of export and axiom clauses that avoids circularity in reasoning[11].)

5.11 Useful lemmas

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

Lemma 1. $\Delta \vdash (\textbf{also}_{\text{i}\in\text{I}}\ \{\text{P}_\text{i}\}\_\{\text{Q}_\text{i}\}) \Longrightarrow \{\text{P}_\text{k}\}\_\{\text{Q}_\text{k}\}$ for all $\text{k} \in \text{I}$.

Lemma 2. If $\Delta \vdash \{\text{P}\}\_\{\text{Q}\} \Longrightarrow \{\text{S}_\text{i}\}\_\{\text{T}_\text{i}\}$ for all $\text{i} \in \text{I}$, then $\Delta \vdash \{\text{P}\}\_\{\text{Q}\} \Longrightarrow (\textbf{also}_{\text{i}\in\text{I}}\ \{\text{S}_\text{i}\}\_\{\text{T}_\text{i}\})$.

For Body Verification:

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

6. Conclusions and related work

The presented proof system supports two complementary mechanisms that can express relationships between abstractions in the logic. Such relationships are pervasive in O-O programs, and facilitate flexible client reasoning, access control, specification inference, and constraints on the implementation of abstractions. Moreover, the system offers a sound way to verify various forms and uses of shared multiple inheritance. By virtue of extending Parkinson and Bierman's system, the examples in [27] illustrate that it can also deal with behavior extension, restriction and modification, as well as representation replacement in subclasses. It is modular and every method body is verified only once. MultiStar implements these features in an automatic tool that, as the Gobo case study shows, holds good promise for verifying real-world software.

[11] The layered assumption structure does not rule out axioms that depend on other axioms. If we want an axiom Q in class C_2 that depends on axiom P in class C_1, we can write the axiom $(\forall \text{x} <: \text{C}_1 \cdot \text{P}[\text{x}/\textbf{Current}]) \Rightarrow \text{Q}$ in class C_2 instead, where x is a fresh variable. After axiom verification, method and statement verification can use $(\forall \text{y} <: \text{C}_2 \cdot \text{Q}[\text{y}/\textbf{Current}])$ where y is fresh. The proof system enables the specification and verification of invariants for aggregate structures involving multiple objects of different types.

We are not aware of any other proof system or tool that can verify our examples and case study. Nevertheless, there are many relationships with other work:

Axiom clauses We do not know of any existing specification mechanisms that are closely related to axiom clauses.

Class invariants form the basis of several O-O specification and verification approaches, including Spec# [1] and JML [19]. Two main flavors of class invariants exist: private invariants, as exemplified by the object invariants of Spec#, and public invariants [18], which include JML's derived invariants and the invariants of Jacobs and Piessens for describing relationships between inspector methods [12]. Class invariants, like axiom clauses, constrain subclasses. However, there are several important differences between them. Class invariants either define exactly when an object is consistent (private invariants), or describe abstract properties of consistent objects (public invariants). Axiom clauses have no notion of object consistency. Class invariants are expected to hold at particular points in a program and may be broken at others, according to the employed invariant protocol [7]. Axiom clauses are true invariants in the sense that they hold everywhere. The relationships they describe cannot be violated by assignment statements, and hence there are no problems with e.g. method callbacks [20, 21]. Class invariants are expressed in terms of fields (including model fields) and pure method calls. Public invariants constrain operations that must establish or preserve them. Private invariants constrain the pure methods they use and indirectly also other operations by depending on the private invariant. Axioms are expressed as logical predicates, and they constrain logical abstractions of data. The verification of class invariants involves the inspection of method bodies (private invariants – for the proof obligations see e.g. [22]), or private invariants and the specifications and/or bodies of pure methods (public invariants). Axiom clauses are verified prior to methods and do not depend on methods in any way. The main differences between class invariants and axiom clauses are summarized in Table 2.

Export clauses There is some correspondence between the class axioms of Kassios [15, 16] and apfs/export clauses. Class axioms can be used to axiomatize the specification and program attributes of a class. In a class implementation, class axioms typically describe abstract state (represented by specification attributes) as a function of the concrete implementation state (represented by program attributes). This corresponds loosely to the way apf entry definitions relate the concrete state to apf arguments. Furthermore, class axioms can describe consequences of a specification attribute such as a class invariant, which typically include framing properties and relationships between other specification attributes. This corresponds somewhat to export clauses that describe properties of apf arguments or relationships between them. The framework of class axioms does not include inheritance, so despite the similar names, it is best to

Class invariants	Axiom clauses
Related to object consistency	No notion of object consistency
Hold at particular program points	Hold everywhere
Expressed i.t.o. fields and pure method calls	Expressed as logical predicates
Constrain operations	Constrain logical abstractions of data
Verification involves methods	Verification cannot involve methods

Table 2. The main differences between class invariants and axiom clauses.

compare them with export and not axiom clauses. Class axioms can also define method specifications/implementations, which apfs and export clauses are incapable of.

In jStar [5], the modifier 'export' can be added to an apf entry definition to expose it to all other classes. Even though our export clauses are more general and flexible than this mechanism, MultiStar supports it as a useful shorthand.

The rules for lossless casting by Chin et al. [4] describe relationships between predicates that provide full and partial views of objects. A view predicate describes the contents of the fields of an object directly: a full view of object o provides full knowledge of all o's fields, while a partial view with respect to class C describes only values of fields introduced by C and its ancestors. View predicates and relationships between them are generated automatically. The relationships do not have to be verified and do not constrain subclasses.

Krishnaswami et al. use so-called 'static specifications' in [17] to specify relationships between abstract predicates. Although not presented in an O-O context, these relationships must be satisfied by implementations and are thus related to our export clauses.

The lemma functions of VeriFast [13] record proofs of relationships between predicates. The relationships are then used in reasoning; the proof of the Composite pattern in [14] provides a good example. Lemma functions, like export clauses, do not constrain subclasses.

Multiple inheritance Surprisingly few systems exist for reasoning about multiple inheritance. The system in [23] also uses separation logic, but without abstraction mechanisms such as apfs. Most of the paper is devoted to elementary separation logic proof rules that also apply in a single-inheritance context. Diamond inheritance is never treated, and the bodies of inherited methods are reverified in subclasses.

The focus of [6] is on behavioral subtyping. It proposes to verify behavioral subtyping of methods lazily, i.e. only to the extent demanded by client code. Supplier code is then continually re-verified as a client's use of it grows.

The restricted form of 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, including Spec# [1] and the JML toolset [2], provide support for specifying and verifying interface inheritance. Both Spec# and JML use pure expressions of the programming language for specification, and follow a class invariant-based approach to verification.

A. Proof system semantics

An outline of the semantics and soundness proof follows. Our system's semantics is similar to that of Parkinson and Bierman's system in [27]. The most interesting difference is the treatment of export and axiom information in the soundness proof of the program verification rule (Theorem 11 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:

$$\begin{array}{rcl} \mathcal{I} & : & \mathsf{Preds} \to (\mathsf{Vals}^* \to \mathcal{P}(\Sigma)) \\ \mathcal{L} & : & \mathsf{Vars} \to \mathsf{Vals} \end{array}$$

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

Under mild syntactic restrictions, obeyed in this paper and detailed in [26], one can show that every set of disjoint predicate definitions is satisfiable:

Lemma 5. For any set of definitions $\mathrm{W}_1, \ldots \mathrm{W}_\mathrm{m}, \mathrm{D}_1, \ldots \mathrm{D}_\mathrm{n}$ where $\mathrm{W_i}$ has form $\mathrm{w_i}(\overline{\mathrm{x_i}}) = \mathrm{Q_i}$ and $\mathrm{D_j}$ is listed in class $\mathrm{G_j}$, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models [\bigwedge_{\mathrm{i}\in 1..\mathrm{m}} \forall \overline{\mathrm{x_i}} \cdot \mathrm{w_i}(\overline{\mathrm{x_i}}) \Leftrightarrow \mathrm{Q_i}] \wedge [\bigwedge_{\mathrm{j}\in 1..\mathrm{n}} \mathit{apf}_{\mathrm{G_j}}(\mathrm{D_j})]$ 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 6. $\mathcal{I} \models_\mathrm{n} \{\mathrm{P}\}\bar{\mathrm{s}}\{\mathrm{Q}\}$ iff whenever $\sigma, \mathcal{I}, \mathcal{L} \models \mathrm{P}$ then $\forall \mathrm{m} \leq \mathrm{n} \cdot$

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

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

Definition 7 (Method verification semantics)**.** If m in G is non-abstract, let $\bar{s}$ denote its body.

$\mathcal{I}, \Gamma \models_0 G.m \mapsto (\overline{x},\{P\}\_\{Q\})$ always holds.
$\mathcal{I}, \Gamma \models_{n+1} G.m \mapsto (\overline{x},\{P\}\_\{Q\})$ iff
$\mathcal{I} \models_n \Gamma \Rightarrow \mathcal{I} \models_{n+1} \{P * \mathbf{Current} : G\}\bar{s}\{Q\}$
if G is non-abstract and true otherwise.

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

$\mathcal{I} \models_n \Gamma$ iff $\forall$methodspec $\in \Gamma \cdot \mathcal{I}, \Gamma \models_n$ methodspec

We next define the semantics of the statement judgement.

Definition 8. $\Delta; \Gamma \models \{P\}\bar{s}\{Q\}$ iff for all $\mathcal{I}$ and n, if $\mathcal{I} \models \Delta$ and $\mathcal{I} \models_n \Gamma$, then $\mathcal{I} \models_{n+1} \{P\}\bar{s}\{Q\}$

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

The judgements are sound with respect to their semantics.

Lemma 9.

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

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

Lemma 10.

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

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

Theorem 11. If a program and its main body $\bar{s}$ can be proved with the program verification rule, then $\forall \mathcal{I}$,n$\cdot$ $\mathcal{I} \models_n \{true\}\bar{s}\{true\}$.
Proof.

1. *The goal*. We have to prove $\forall \mathcal{I}$,n$\cdot$ $\mathcal{I} \models_n \{true\}\bar{s}\{true\}$, which abbreviates $\forall \mathcal{I}$,n$\cdot$ whenever σ, $\mathcal{I}$, $\mathcal{L} \models$ true, then $\forall m \leq n \cdot$ 1) σ,$\bar{s} \longrightarrow^{m}$ fault does not hold, and 2) if σ,$\bar{s} \longrightarrow^{m} \sigma'$,$\epsilon$ then σ', $\mathcal{I}$, $\mathcal{L} \models$ true. This can be simplified to $\forall n \cdot \sigma,\bar{s} \longrightarrow^{n}$ fault does not hold.
2. *Strengthened assumptions*. Let $\Delta_T \stackrel{def}{=} \bigwedge_{i \in 1..t}$ *apf*(L_i), where $L_1 \ldots L_t$ are all classes in the program. By Lemma 10, we can strengthen the assumptions under which all classes and the main body have been verified. For every class L_i, we have Δ_T, Δ_E, Δ_A; $\Gamma \vdash_c L_i$, and $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \vdash_s \{true\}\bar{s}\{true\}$ also holds for the main body $\bar{s}$.
3. *The interpretation* $\mathcal{I}'$. Since $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \vdash_s \{true\}\bar{s}\{true\}$, Lemma 9 guarantees $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \models \{true\}\bar{s}\{true\}$. This abbreviates $\forall \mathcal{I}$,n$\cdot$ if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_n \Gamma$, then $\mathcal{I} \models_{n+1} \{true\}\bar{s}\{true\}$, which can be simplified to $\forall \mathcal{I}$,n$\cdot$ if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_n \Gamma$, then $\forall m \leq n + 1 \cdot \sigma,\bar{s} \longrightarrow^{m}$ 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 $\forall n \cdot \mathcal{I}' \models_n \Gamma$, then we can instantiate $\mathcal{I}$ to $\mathcal{I}'$ in the formula and simplify to obtain $\forall n \cdot \sigma,\bar{s} \longrightarrow^{n}$ fault does not hold. Therefore $\mathcal{I}'$ serves as a witness that $\bar{s}$ will never fault, which is exactly our goal.
Let $\mathcal{I}'$ be the interpretation whose existence is guaranteed by Lemma 5 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** $\{w_1(\overline{x_1}) = Q_1; \ldots; w_n(\overline{x_n}) = Q_n\}$ in class L. Since *apf*(L) $\vdash_e$ E, we know [*apf*(L) $\wedge$ ($\bigwedge_{i \in 1..n} \forall \overline{x_i} \cdot w_i(\overline{x_i}) \Leftrightarrow Q_i$)] $\Rightarrow$ P. The interpretation $\mathcal{I}'$ satisfies the antecedent, so we also have $\mathcal{I}' \models$ 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 $H_1 \ldots H_k$, let *level*(G) $\stackrel{def}{=}$ 1 + max(0, *level*(H_1), …, *level*(H_k)). Furthermore, *P*(n) $\stackrel{def}{=} \forall$ G in the program such that *level*(G) $\leq$ n and for all axiom clauses a: P in the listing of G, ($\Delta_T \wedge \Delta_E$) $\Rightarrow$ *axiominfo*(G, a: P).

   - Base case. Consider an arbitrary class G with *level*(G) ≤ 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*(G, 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 direct instances of G. Since ($\Delta_T \wedge \Delta_E \wedge$ **Current** : G) $\Rightarrow$ P by the Implication premise, we therefore also know ($\Delta_T \wedge \Delta_E$) $\Rightarrow$ *axiominfo*(G, a: P).

     Thus *P*(1) holds.
   - Step case. Suppose *P*(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*(H, a: Q), and by the Parent Consistency premise of a: Q we know ($\Delta_T \wedge \Delta_E \wedge$ Q) $\Rightarrow$ P. Therefore ($\Delta_T \wedge \Delta_E$) $\Rightarrow$ *axiominfo*(H, a: P). We have ($\Delta_T \wedge \Delta_E$) $\Rightarrow$ *axiominfo*(G, a: P) if G is abstract, and the same holds if G is non-abstract since

the Implication premise of a: P guarantees ($\Delta_T \wedge \Delta_E \wedge$ $\mathbf{Current}$: G) $\Rightarrow$ P. Thus *P*(n+1) holds.

So $\mathcal{I}' \models \Delta_T \wedge \Delta_E \wedge \Delta_A$.

6. *Wrapping up*. We still have to prove $\forall \mathrm{n} \cdot \mathcal{I}' \models_{\mathrm{n}} \Gamma$. Let m be an arbitrary method in the program. Since $\Delta_T \wedge \Delta_E \wedge \Delta_A; \Gamma \vdash_m \mathrm{m}$, by Lemma 9 we know for all n and every spec of m that $\mathcal{I}'$, $\Gamma \models_{\mathrm{n}}$ spec. Thus $\forall \mathrm{n} \cdot \forall \mathrm{methodspec} \in \Gamma \cdot \mathcal{I}'$, $\Gamma \models_{\mathrm{n}}$ methodspec, in other words $\forall \mathrm{n} \cdot \mathcal{I}' \models_{\mathrm{n}} \Gamma$. □

Acknowledgments

Special thanks to Matthew Parkinson, Peter O'Hearn, Bertrand Meyer, Sebastian Nanz, Carlo Furia and Martin Nordio for feedback on early versions of this work. Matthew Parkinson also helped with the implementation of MultiStar's backend. Van Staden is supported by ETH Research Grant ETH-15 10-1. Calcagno is partially funded by EPSRC.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS '05*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[3] L. Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76(2-3):138–164, 1988.

[4] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 87–99, New York, NY, USA, 2008. ACM.

[5] D. Distefano and M. J. Parkinson J. jStar: towards practical verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM.

[6] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning for multiple inheritance. In *IFM '09*, pages 215–230, Berlin, Heidelberg, 2009. Springer.

[7] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In J. Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.

[8] ECMA International. *Standard ECMA-367. Eiffel: Analysis, Design and Programming Language*. 2nd edition, June 2006.

[9] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[10] EVE. The Eiffel Verification Environment. `http://eve.origo.ethz.ch/`.

[11] Gobosoft. The Gobo Eiffel Structure Library. `http://www.gobosoft.com/eiffel/gobo/structure/index.html`.

[12] B. Jacobs and F. Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5):55–75, June 2007.

[13] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, August 2008.

[14] B. Jacobs, J. Smans, and F. Piessens. Verifying the composite pattern using separation logic. *SAVCBS Composite pattern challenge track*, 2008.

[15] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[16] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 2010. To appear.

[17] N. R. Krishnaswami, L. Birkedal, J. Aldrich, and J. C. Reynolds. Idealized ML and Its Separation Logic. Draft available online at `http://www.cs.cmu.edu/~neelk/idealized-ml-draft.pdf`. 2006.

[18] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE Computer Society, 2007.

[19] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[20] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[21] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[22] K. R. M. Leino and W. Schulte. A verifying compiler for a multi-threaded object-oriented language. *Software System Reliability and Security*, 9:351–416, 2007.

[23] C. Luo and S. Qin. Separation logic for multiple inheritance. *Electr. Notes Theor. Comput. Sci.*, 212:27–40, 2008.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[25] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05: Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 2005. ACM.

[26] M. J. Parkinson. Local reasoning for Java. PhD thesis. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, November 2005.

[27] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–86, New York, NY, USA, 2008. ACM.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.