Report

Reasoning about multiple related abstractions with MultiStar

Author(s):
Staden, Stephan van; Calcagno, Cristiano

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006851133

Rights / License:
In Copyright - Non-Commercial Use Permitted
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]: Program Verification; D.3.3 [Programming Languages]: Language Constructs and Features—Classes and inheritance

General Terms Languages, Theory, Verification, Tools

Keywords Separation logic, Multiple abstractions, Multiple inheritance

1. Introduction
The use of data abstractions is a hallmark of object-oriented (O-O) programming. A class is probably the first 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 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 [22, 24]. An apf (abstract predicate family) P provides 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, afps $S$ and $P$ can be used to provide an abstraction of students and persons in a program respectively. Class STUDENT can define the entries $S_{\text{STUDENT}}$ and $P_{\text{STUDENT}}$, 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

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

In other words, the dynamic type of the first argument of an apf predicate determines which apf entry applies. Apf predicates can therefore be seen to mirror dynamic dispatch of O-O programs in the logic. Apfs in the logic are 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:

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

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

$$x.S(age: a, exm: e) \iff \{x.P(age: a) \ast x.RestStoP(exm: e)\} \quad (A)$$

where $\text{RestStoP}$ 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:

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

// Manipulation of ‘person’ abstraction by library routines.

$$\{x.P(age: a+1)\}$$
$$\{x.P(age: a+1) \ast x.RestStoP(exm: e)\}$$
$$\{x.S(age: a+1, exm: e)\}$$

The Frame rule of separation logic guarantees that the disjoint $x.RestStoP(exm: e)$ remains unchanged. In essence, the client uses the property 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**

$$\forall x,a,e: x:S_{\text{STUDENT}} \Rightarrow \{x.S(age: a, exm: e) \iff \{x.P(age: a) \ast x.RestStoP(exm: e)\}\}$$

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**

$$S_P: \forall a,e: S_{\text{STUDENT}} \iff \{x.S(age: a, exm: e) \iff \{x.P(age: a) \ast x.RestStoP(exm: e)\}\}$$

then clients can use the stronger implication

$$\forall x,a,e: x:S_{\text{STUDENT}} \Rightarrow \{x.S(age: a, exm: e) \iff \{x.P(age: a) \ast x.RestStoP(exm: e)\}\}$$

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.

Axioms 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**

$$\text{exm\_non\_neg}: \forall a,e: S_{\text{STUDENT}}(age: a, exm: e) \iff 0 \leq e$$

$$S_{\text{constraint}}: \forall a,e: S_{\text{STUDENT}}(age: a, exm: e) \Rightarrow S_{\text{STUDENT}}(age: a, exm: e)$$

Axiom clause ‘exm\_non\_neg’ guarantees clients that $0 \leq e$ whenever they know $x.S(age: a, exm: e)$ and $x:S_{\text{STUDENT}}$. Subclass representation constraints such as the one expressed in the ‘$S_{\text{constraint}}$’ 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 [24] with export/axiom clauses, abstract classes, abstract methods and shared multiple inheritance2 where fields and methods of common ancestor classes are not replicated in the descendant [9]. Apart from the use of apfs to support abstraction

---

1 Matthew Parkinson pointed out in private communication that the informal discussion on representation constraints right before Section 5.1 and in Section 5.5 of [24] can be made rigorous by using export and axiom clauses.

2 Inheritance with virtual base classes in C++ terminology [10].
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. 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 [6] 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 iteratory hierarchy of Figure 1. The complete code and specifications of the examples and Gobo case study are available online [11].

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 [9]. 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 static and dynamic specifications, written in pre-post form \[ \lbrack P \rbrack \{Q\}, \{P_1\}, \{Q_1\}, \text{ also } \{P_2\}, \{Q_2\} \] to indicate that both are satisfied. 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 \( x.C::m(\sigma) \) in C++ style.

We omit the target of a method call or field assignment if it is Current. Similarly in the logic, \( f \mapsto e \) abbreviates Current. \( f \mapsto e \), 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 [24]: 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 \( p(t; e) \) is replaced with \( p_C(t; e) \). 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 [24] 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 information without introducing new predicate families.

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.

For the constructor we have to prove that its body satisfies the static specification (note that a Precursor call is a direct call):
class CELL
define x.CellCELL(val: v) as x.value val

feature
    introduce CELL(v: int)
dynamic {value v, .} {Cell(val: v)}
do value v

    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.CCOUNTER(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

counter: int

end

Figure 2. The CELL and COUNTER classes.

\[
\text{Cell}(v) \rightarrow \text{Cell}(v, c) \quad \text{Cn}(v) \rightarrow \text{Cn}(v, c)
\]

Figure 3. The CCELL class and two library methods.

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 CnCell(cnt: c)) and then the rule of Consequence, we can derive each also-ed static specification, which is sufficient to conclude the proof\(^4\). Another proof obligation for value is Behavioral subtyping, where

\(^3\) 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.

\(^4\) By Lemma 2 on page 12.

---

\[\forall v \in \text{CELL} \mapsto [\forall c, v \in \text{CnCell}(v, c) \mapsto x.CellCell(v, c) \mapsto x.CnCell(v, c)] \mapsto x.CellCellCell(v, c) \mapsto x.CnCellCell(v, c)\]

Another application of Consequence proves the other also-ed static spec and completes the proof\(^5\). As the body operates on state described by CellCELL and CnCOUNTER, the proof obligations and separation logic’s faulting semantics

\(^5\) By Lemma 1 on page 12.

\(^6\) By Lemma 3 on page 12.
demand that we 'grow' Cell\textsubscript{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:

\begin{verbatim}
{true} cc := new CCELL(5); use_counter(cc) \{ cc.Cell(val: 5, cnt: 10) \}
{true} cc := new CCELL(5); use_cell(cc,20) \{ cc.Cell(val: 20, cnt: 2) \}
\end{verbatim}

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

\begin{verbatim}
{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: 1) \}
\{ cc : CCELL \& cc.Cell(val: 5) \}
use_cell(cc,20)
\{ cc : CCELL \& cc.Cell(val: 20) \}
\{ cc : CCELL \& cc.Cell(val: 20, cnt: 1) \}
\{ cc.Cell(val: 20, cnt: 2) \}
\end{verbatim}

Information about cnt is lost in the postcondition, which is sound because use\_cell could call set\_value more than once. In a version of CCELL where Cn\textsubscript{CCELL} is defined to include the Cell\textsubscript{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.

### 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:

\begin{verbatim}
\{cc2.Cell(val: v, cnt: c)\}
\{cc2.Cn(cnt: c)\}
cc2.increment()
\{cc2.Cn(cnt: c+1)\}
???\}
\{cc2.set\_value(10)\}
\{cc2.Cell(val: _, cnt: _)\}
\end{verbatim}

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 complicates matters because common ancestor fields are shared. This is unproblematic for our proof system, although 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, we require that if clauses with the same name are present in a subclass, then it must all be identical. An axiom clause copied in this way 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 multi-
class PERSON
define x.PPERSON(age: a) as x.age := a
export ∀x,a - x.PPERSON(age: a) ⇔ x.age := a where {}

feature
  introduce PERSON(a: int)
dynamic {age := a} \{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: a)} \{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.PSTUDENT(age: a) as x.PPERSON(age: a)
x.STUDENT(age: a, exam: e) as x.PSTUDENT(age: a) * x.exams := e
x.RestSToPSTUDENT(exm: e) as x.exams := e
export ∀x,a,e - x.PSTUDENT.age: int, e: int, where {}
axiom S\{P(age: a)\}

feature
  introduce STUDENT(a: int; e: int)
dynamic {age := a, exams := e} \{S(age: a, exam: e)\}
do Precursor{PERSON}(a); exams := e end

introduce take_exam()
static {S(age: a, exam: e)} \{S(age: a, exam: e+1)\}
do tmp: int; tmp := exams; exams := tmp+1 end

exams: int end

Figure 5. The PERSON and STUDENT classes.

Figure 6. The SMUSICIAN class.

class SMUSICIAN inherit STUDENT MUSICIAN define
x.PSMUSICIAN(age: a) as x.PPERSON(age: a)
x.PSMUSICIAN(age: a, exam: e) as x.PSTUDENT(age: a, exam: e)
x.PSMUSICIAN(age: a, pfm: p) as x.MUSICIAN(age: a, pfm: p)
x.PSMUSICIAN(age: a, exam: e, pfm: p) as x.PPERSON(age: a) *
x.RestSToPSTUDENT(exm: e) * x.RestMtoPSTUDENT(exm: e)
x.RestSToPSTUDENT(exm: e) as x.RestSToPSTUDENT(exm: e)
x.RestMtoPSTUDENT(exm: e) as x.RestMtoPSTUDENT(exm: e)
x.RestMtoPSMUSICIAN(pf: p) as x.RestMtoPSMUSICIAN(pf: p)
x.RestMtoPSMUSICIAN(pf: p) as x.RestMtoPSTUDENT(exm: e)
x.RestMtoPSMUSICIAN(exm: e) as x.RestMtoPSTUDENT(exm: e)

axiom
SM_S: ∀a,e,p - SM(age: a, exam: e, pfm: p) ⇔
[S(age: a, exam: e) * RestSMtoPSTUDENT(exm: e)]
SM_M: ∀a,e,p - SM(age: a, exam: e, pfm: p) ⇔
[M(age: a, exam: e) * RestMtoPSTUDENT(exm: e)]

feature
  introduce SMUSICIAN(a: int; e: int; p: int)
dynamic {age := a, exams := e, performances := e} \{SM(age: a, exam: e, pfm: p)\}
do Precursor{PERSON}(a,e); Precursor{MUSICIAN}(a,p) end

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

end
dynamic

SM_S: ∀a,e,p - SM(age: a, exam: e, pfm: p) ⇔
[S(age: a, exam: e) * RestSMtoPSTUDENT(exm: e)]

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 our 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:

\[
\{age := a, exams := e, performances := e\} \\
\text{Precursor}\{PERSON\}(a,e) \\
\{S_{STUDENT}(age: a, exam: e) * performances := e\} \\
\{P_{PERSON}(age: a) * RestSToPSTUDENT(exm: e) * performances := e\} \\
\{age := a, RestSToPSTUDENT(exm: e) * performances := e\} \\
\text{Precursor}\{MUSICIAN\}(a,p) \\
\{M_{MUSICIAN}(age: a, pfm: p) * RestSToPSTUDENT(exm: e)\} \\
\{P_{PERSON}(age: a) * RestMtoPSTUDENT(exm: e) * RestSToPSTUDENT(exm: e)\} \\
\{SM_{SMUSICIAN}(age: a, exam: e, pfm: p)\}
\]

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:

\[
\begin{align*}
\{ & \text{SM(age: a, exam: e, pfm: p)} \} \\
\{ & \text{S(age: a, exam: e)} \} \to \text{RestSMtoS(pfm: p)} \\
\{ & \text{perform()} \} \\
\{ & \text{SM(age: a, exam: e+1, pfm: p)} \} \\
\{ & \text{M(age: a, pfm: p)} \} \to \text{RestSMtoM(exam: e+1)} \\
\{ & \text{perform()} \} \\
\{ & \text{SM(age: a, exam: e+1, pfm: p+1)} \} \\
\{ & \text{M(age: a, exam: e+1)} \} \to \text{RestSMtoM(exam: e+1)} \\
\end{align*}
\]

The specification overhead incurred by using axioms 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. These inferred specifications are guaranteed to be implemented by all subclasses, and no dynamic type information is needed to use them. Yet the system is still flexible – a subclass can always choose to satisfy such constraints vacuously by defining selected afp entries as false. Class CCell in [24] 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 [6] 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 [11]. 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 afp 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

\[
\text{dynamic \{ \{true\} \{Cc(val:v, cnt:0)\} }
\]

instead of

\[
\text{dynamic \{ \{value \to \_, \_ \} \{Cc(val:v, cnt:0)\} }
\]

In detail, the front-end:

1. Uses jStar’s new statement, whose specification is given by the triple \{true\}x := new C\{x : C\}. This allows us to omit the fields in the dynamic precondition\(^7\).

2. Adds all fields (including ancestor ones) to the static precondition when checking Body verification, and consumes 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 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 [10].

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

Background theory used by the jStar theorem prover is encoded as a list of sequent rules. A sequent is of the form \(P \mid Q \vdash R\), meaning \((P \ast Q) \Rightarrow (P \ast R)\). Each sequent rule has the form

\[
A \mid B \vdash C \\
\text{if} \\
D \mid E \vdash F
\]

If the prover is trying to prove a sequent that matches the rule’s conclusion \(A \mid B \vdash C\), it suffices to prove the sequent where the rule’s premise \(D \mid E \vdash F\) replaces the matched predicates. A new proof goal is thus obtained, and the proof is complete when the goal is of the form \(G \mid H \vdash \). For details the reader is referred to [6].

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:

\[
\begin{align*}
| w(x) \vdash & \text{if} \\
| P \vdash & \text{if}
\end{align*}
\]

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 \(P \Rightarrow (Q_1 \ast \ldots \ast Q_n)\) is added to the background theory as a set of \(n\) rules, where rule \(i \in 1..n\) has the form
| P ⊢ Q_i, if Q_i | Q_1 * ... * Q_{i-1} * Q_{i+1} * ... * Q_n ⊢ |

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 P ⇒ (Q_1 * ... * Q_n) written in class C is encoded as n rules, with rule i ∈ 1..n of the form

| P ⊢ Q_i, if Q_i | Q_1 * ... * Q_{i-1} * Q_{i+1} * ... * Q_n ⊢ x <: 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:

- 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 [12] 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 total 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

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

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.

| 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 |
| do |
| v: G; w: G; |
| v := item(); w := other.item(); |
| replace(w); replace(v) |
| end |

Figure 7. A simplified extract of DYNAMIC_CURSOR
the axiom clauses that iterators demand. Consider for example the simplified extract of DYNAMIC_CURSOR in Figure 7. 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 verication 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, i: c1, content: c1) * 
ItemAt(res: r2, ref: iter2, i: c2, content: c1) * 
Replaced(ref: iter1, value: r2, newcontent: c2, oldcontent: c1, iter: i) * 
Replaced(ref: iter2, value: r1, newcontent: c1, oldcontent: c2, iter: i)]
⇒ Swapped(ref1: iter1, ref2: iter2, iter: i, oldcontent: c1, newcontent: c3)

This invariant property relates the ItemAt, Replaced and Swapped abstractions, and illustrates the usefulness of axiom clauses as a general specication mechanism.

The complete specications and code of the case study are included in the MultiStar download [11].

5. Formalization
This section contains a formal treatment of the programming language with specications and its proof system. The language features abstract classes and multiple inheritance. Export and axiom specications are supported. The proof system is based on the one of Parkinson and Bierman in [24]. For space reasons we focus mostly on the new extensions.

5.1 Language syntax
The grammar of our kernel language with multiple inheritance and specications is shown in Figure 8. A sequence of c’s is denoted by τ. 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 eld shadowing, for instance, and guarantees that methods or elds 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-structor methods present in its parents (in MultiStar and the examples, specication shorthand are employed to achieve this). The shared semantics of multiple inheritance is used, which is popular in Eiffel [9] and known as inheritance with virtual base classes in C++ [10]. Common ancestor elds 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 in the same in subclasses. This simplies 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 [10]).

The operational semantics is therefore similar to e.g. Java’s and omitted. Conigurations 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 specications and proofs have the following grammar.

\[ P, \ Q, \ S, \ T, \ Δ := ∀ x : P | \ P \equiv \ Q | \ false | e = e' | x : G | x \leftarrow G | x.f := e | P \star Q | x.g(\tau, v) \]

Afp predicate

\[ L := A b \ class \ G \ inherit \ H \ define \ \overrightarrow{F} \ export \ \overrightarrow{X} \ axiom \ \overrightarrow{M} \ feature \ \overrightarrow{F} \ \overrightarrow{X} \ end \]

Ab := abstract | ∃
D := x.p.c(\overrightarrow{y}) as P
E := P where \{ \overrightarrow{W} \}
W := w(\overrightarrow{X}) = P
A := ∈ : P
M := introduce m(\overrightarrow{Args}) Rt Sd Ss B
| override m(\overrightarrow{Args}) Rt Sd Ss B
| inherit m(\overrightarrow{Args}) Rt Sd Ss
| introduce abstract m(\overrightarrow{Args}) Rt Sd
| inherit abstract m(\overrightarrow{Args}) Rt Sd
F := f : Type
Sd := dynamic Spec
Ss := static Spec
Spec := \{ P \}, \{ Q \} | \{ P \}, \{ Q \} also Spec
B := do \{ end
s := x : Type
\begin{align*}
& | x := e \\
& | x := y.f \\
& \text{x.f} := e \\
& | x := y.m(\overrightarrow{τ}) | y.m(\overrightarrow{τ}) \\
& | x := y.G.m(\overrightarrow{τ}) | y.G.m(\overrightarrow{τ}) \\
& | x := \text{new} G
\end{align*}

e := x | e + e | e = e | \text{Void} | 0 | 1 | 2 | \ldots
Type := int | bool | G
Args := x := f : Type
Rt := ∈ : Type
Field declaration
Dynamic specication
Static specication
Specification
Method body
Local variable declaration
Assignment
Field lookup
Field assignment
Dynamically dispatched call
Direct method call
Object allocation
Expression
Formal arguments
Return type

Figure 8. The kernel language grammar.
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 \( x \neq \text{Void} \), and \( x : G \Rightarrow x : G \) holds. Within a context, if \( x \) is declared of type \( G \) then \( x : G \) whenever \( x \neq \text{Void} \).

The first argument of an apf predicate or entry is written as a prefix. For apf predicates it is never \text{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. For an in-depth treatment of apfs, the reader is referred to [24] for lack of space.

Other predicates have the usual intuitionistic separation logic semantics. Informally the predicate \( x.f \) holds for disjoint portions of the heap. Readers are asked to view the f field of object \( x \) has value \( e \), and \( P \) holds. The quantifier is placed in the innermost position. \( FV(P) \) denotes the free variables of \( P \); every method precondition \( P \) must satisfy \textbf{Result} \( \notin FV(P) \).

In the rest of the formalization, the symbols \( P, Q, S \) and \( T \) are used for assertions and predicates, and \( \Delta \) for assumptions.

### 5.4 Specification refinement

We expand on Parkinson and Bierman’s formalization of specification refinement in [24]. If the specification \( \{ P_1 \}.\{ Q_1 \} \) is refined by \( \{ P_2 \}.\{ Q_2 \} \), then any \( \pi \) which satisfies \( \{ P_1 \}.\{ Q_1 \} \) also satisfies \( \{ P_2 \}.\{ Q_2 \} \). In this case we write \( \Delta \vdash \{ P_1 \}.\{ Q_1 \} \Rightarrow \{ P_2 \}.\{ Q_2 \} \), which denotes the existence of a proof tree with leaves \( \Delta \vdash \{ P_1 \}.\{ Q_1 \} \) and root \( \Delta \vdash \{ P_2 \}.\{ Q_2 \} \) built with the structural rules of separation logic (Consequence, Frame, Auxiliary Variable Elimination, Disjunction, and others). In the context of method specification refinement, the Consequence and Frame rules are given by:

\[
\begin{align*}
\Delta \Rightarrow (P' \Rightarrow P) & \vdash \{ P \}.\{ Q \} \Rightarrow \{ Q' \}.\{ Q' \} & \text{Consequence} \\
\Delta \vdash \{ P' \}.\{ Q' \} & \vdash \{ P \}.\{ Q \} & \text{Frame}
\end{align*}
\]

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

Method specifications can be combined with \textbf{also} (Definition 1 in [24]):

\[
\begin{align*}
\{ P_1 \}.\{ Q_1 \} \ \textbf{also} \ \{ P_2 \}.\{ Q_2 \} \quad & \text{def} \\
\{(P_1 \land x = 1) \lor (P_2 \land x \neq 1)\} \Leftarrow \{(Q_1 \land x = 1) \lor (Q_2 \land x \neq 1)\}
\end{align*}
\]

where \( x \) denotes a fresh auxiliary variable. The specifications \( \{ P_1 \}_G\{ Q_1 \} \) and \( \{ P_2 \}_G\{ Q_2 \} \) are equivalent w.r.t. \( \Delta \) iff both \( \Delta \vdash \{ P_1 \}.\{ Q_1 \} \Rightarrow \{ P_2 \}.\{ Q_2 \} \) and \( \Delta \vdash \{ P_2 \}.\{ Q_2 \} \Rightarrow \{ P_1 \}.\{ Q_1 \} \). Two specifications are equivalent iff they are equivalent w.r.t. all \( \Delta \). It can be shown that also is commutative, associative and idempotent modulo equivalence with identity \( \{ \text{false} \}.\{ \text{true} \} \). The notation \textbf{also} \( i \in I \) \( \{ P_i \}_G\{ Q_i \} \) denotes the specification \( \{ P_1 \}_G\{ Q_1 \} \) \textbf{also} \( \ldots \) \textbf{also} \( \{ P_m \}_G\{ Q_m \} \), where \( e_1 \ldots e_m \) are the elements of set \( I \). Furthermore, when \( I \) is the empty set:

\[
\textbf{also} \ i \in I \ \{ P_i \}_G\{ Q_i \} \quad \text{def} \quad \{ \text{false} \}.\{ \text{true} \}
\]

It always holds that \( \Delta \vdash \{ P \}_G\{ Q \} \Rightarrow \{ \text{false} \}.\{ \text{true} \} \). Other useful lemmas involving \textbf{also} are given in Section 5.11. Finally, we use the abbreviation

\[
\begin{align*}
\Delta \vdash \{ P_1 \}.\{ Q_1 \} \quad & \text{Current} : G \vdash \{ P_2 \}.\{ Q_2 \} \\
\vdash \{ P_1 \}_G \quad & \text{def} \quad \Delta \vdash \{ P_1 \}.\{ Q_1 \} \Rightarrow \{ P_2 \. \text{Current} : G \}.\{ Q_2 \}
\end{align*}
\]

### 5.5 The specification environment

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

\[
\begin{align*}
\Gamma ::= \text{G.a} \rightarrow P & \quad \text{Axiom specification} \\
\text{G.m} \rightarrow (\pi_1.(P_1).\{ Q_1 \}) & \quad \text{Method dynamic specification} \\
\text{G.m} \rightarrow (\pi_1.(S).\{ T \}) & \quad \text{Method static specification} \\
\Gamma & \quad \text{Triples}
\end{align*}
\]

The \( \pi \) in a specification of \( m \) denote its formal argument names. \( \Gamma \) is guaranteed to be a partial function for well-typed programs, and we write \( \Gamma(G.a) = P \) for \( G.a \Rightarrow P \in \Gamma \), etc.

#### 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 \text{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 \text{where}, the assumptions about a class must imply exported information. This is captured by the following proof rule:

\[
\begin{align*}
\Delta \land (\forall x_1 \neg \text{w_1(x_1)} \Rightarrow \{ Q_1 \} \land \ldots \land (\forall x_m \neg \text{w_m(x_1)} \Rightarrow \{ Q_m \})) & \Rightarrow P \\
\Delta \vdash \pi \text{w} \text{where} \{ w_1(x_1) = Q_1 ; \ldots ; w_m(x_1) = Q_m \}
\end{align*}
\]

### 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 method bodies. 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).
5.8 Statement verification

The assumptions \( \Delta \) 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. [23, 24]). For allocation:

\[
\Delta; \Gamma \vdash \{ \text{new} \} \ x : \ A \\
\{ \text{new} \} \ x \rightarrow \ \Gamma \vdash \{ \text{new} \} \ x : \ A
\]

where \( \text{newfields}(G) \) denotes the set of field names listed in \( G \) and all its ancestors.

Dynamically dispatched calls use the dynamic specs of methods in \( \Gamma \), while direct calls use the static ones. Provided \( x \) is not \( y \) and \( x \) is not free in \( \pi \), the rules for result-returning calls are:

\[
\begin{align*}
\Delta; \Gamma \vdash \{ \text{new} \} \ x : \ A \rightarrow \ \Gamma \vdash \{ \text{new} \} \ x : \ A \\
\{ \text{new} \} \ x \rightarrow \ \Gamma \vdash \{ \text{new} \} \ x : \ A
\end{align*}
\]

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

\[
\Delta; \Gamma \vdash \{ \pi \} \ P \quad \Delta \vdash \{ \pi \} \ P \rightarrow Q \\
\Delta; \Gamma \vdash \{ \pi \} \ P \rightarrow Q
\]

The rule of Consequence allows the use of assumptions \( \Delta \):

\[
\Delta \vdash \{ \pi \} \ P \rightarrow Q \\
\Delta; \Gamma \vdash \{ \pi \} \ P \rightarrow Q
\]

5.9 Method verification

The rules for method verification in [24] are extended here to the multiple inheritance case. As for statement verification, the assumptions \( \Delta \) used to verify method definitions contain information about the method’s enclosing class as well as export and axiom information from all other classes.

The rule for method introduction requires no modification for multiple inheritance. A newly introduced method’s static and dynamic specifications must be consistent 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.

\[
\begin{align*}
\Delta, \Gamma \vdash \text{do } \ m \end{align*}
\]

\[
\begin{align*}
\text{B} = \text{do } \ S \\
\text{Sd} = \text{dynamic} \ \{ P_G \} \{ Q_G \} \\
\text{Ss} = \text{static} \ \{ S_G \} \{ T_G \} \\
\Delta \vdash \{ S_G \} \{ T_G \} \quad \Delta, \Gamma \vdash \{ S_G \} \{ T_G \} \rightarrow \{ Q_G \} \{ D \} \\
\Delta; \Gamma \vdash \{ S_G \} \{ T_G \} \rightarrow \{ Q_G \} \{ D \}
\end{align*}
\]

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

\[
\begin{align*}
\Delta; \Gamma \vdash \text{introduce abstract } \ m \text{(Args) Rt Sd Ss B} \end{align*}
\]

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 \( H_1 \) \( \ldots \ H_n \) are the immediate superclasses of \( G \).

\[
\begin{align*}
\Delta, \Gamma \vdash \text{introduce abstract } \ m \text{(Args) Rt Sd Ss B} \end{align*}
\]

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.

\[
\begin{align*}
\Delta, \Gamma \vdash \text{inherit } \ m \text{(Args) Rt Sd Ss B} \end{align*}
\]

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.

\[
\begin{align*}
\Delta, \Gamma \vdash \text{inherit abstract } \ m \text{(Args) Rt Sd Ss B} \end{align*}
\]
5.10 Class and program verification

For class verification, different assumptions are used to verify the various class sections. The formula $\Delta_{APF}$ contains class-specific information and is used to verify export clauses. The assumptions $\Delta_E$ contain export information from all classes, and are used together with $\Delta_{APF}$ to verify axioms. The formula $\Delta_A$ contains axiom information of all classes, and is used with $\Delta_{APF}$ and $\Delta_E$ in method definition verification.

$$\forall E_i \in \mathcal{E} : \Delta_{APF} \vdash E_i$$
$$\forall A_i \in \mathcal{A} : \Delta_{APF} \land \Delta_E \vdash A_i \text{ in Ab G parents } \Pi$$
$$\forall M_i \in \mathcal{M} : (\Delta_{APF} \land \Delta_E \land \Delta_A) ; \Gamma \vdash M_i \text{ in Ab G parents } \Pi$$

$\Delta_{APF}, \Delta_A, \Gamma \vdash c$

Ab class G inherit $\Pi$ define $\mathcal{T}$ export $\mathcal{E}$ axiom $\mathcal{A}$ feature $\mathcal{M}$ end

Finally, here is the rule for program verification:

$$\forall i \in 1..n : \text{ L_i = ...class G_i ...export name axiom A_i feature ...end}$$
$$\Delta_E = \bigwedge_{i \in 1..n} \bigwedge_{A_i \in \mathcal{A}} \mathcal{E}(\text{exportinfo}(V_i))$$
$$\Delta_A = \bigwedge_{i \in 1..n} \bigwedge_{A_i \in \mathcal{A}} \mathcal{A}(\text{axiominfo}(G_i, A_i))$$
$$\Gamma = \text{spec}(L_1 \ldots L_n)$$
$$\forall i \in 1..n : \text{ apf}(L_i), \Delta_E, \Delta_A ; \Gamma \vdash c \quad L_i$$

For class verification, different assumptions are used to verify the various class sections. The formula $\Delta_{APF}$ contains axiom information of all classes, and is used with $\Delta_{APF}$ and $\Delta_E$ in method definition verification.

5.11 Useful lemmas

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

Lemma 1. $\Delta \vdash (\text{also}_{e \in 1} \{P_1\} - \{Q_1\}) \implies \{P_k\} - \{Q_k\}$ for all $k \in I$.

Lemma 2. If $\Delta \vdash \{P\} - \{Q\} \implies \{S_i\} - \{T_i\}$ for all $i \in I$, then $\Delta \vdash \{P\} - \{Q\} \implies (\text{also}_{e \in 1} \{S_i\} - \{T_i\})$.

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 [24] 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.

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\(^6\) form the basis of several O-O verification approaches, including Spec# [2] and JML [16]. Class invariants, like axiom clauses, constrain subclasses. However, they do not constrain data abstractions but rather operations or behaviors: they express consistency conditions of an object that methods must respect [1]. In a basic scheme they are conjoined to constructor postconditions and public method pre- and postconditions, but many other protocols exist [8]. Thus class invariants are expected to hold at particular points in a program and may be broken at others. Our axiom clauses have nothing to do with consistent and inconsistent object states, and are true invariants in the sense that they hold everywhere. Consequently there are no problems with e.g. method callbacks [17, 18]. Class invariants express relationships between methods in terms of method calls. For example, a class invariant of a collection class could be empty() \(=\) (size() = 0), which relates the empty and size methods and is expressed in terms of calls to them. Our axiom clauses specify relationships between logical, instead of operational, abstractions, and are expressed as predicates. Class invariants are verified by analyzing method bodies [19]. Axiom clauses, on the other hand, are verified prior to methods and

\(\text{Also called object invariants.}\)
Axiom clauses

Constrain logical abstractions of data
No notion of object consistency
Hold everywhere
Expressed as logical predicates

Constrain operations
Denote consistent object states
Hold at particular program points
Expressed i.t.o. method calls
Verified together with methods

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

without inspecting any code. The main differences between class invariants and axiom clauses are summarized in Table 2.

Export clauses The rules for lossless casting by Chin et al. [5] 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 [15] 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 [20] 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 [7] 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# [2] and the JML toolset [3], 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 [24]. 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 I, and an interpretation of logical variables L. The interpretation I maps predicate names to their definitions, whereas a definition maps a list of arguments to a set of states:

\[ I : \text{Preds} \rightarrow (\text{Vars}^* \rightarrow \mathcal{P}(\Sigma)) \]

\[ L : \text{Vars} \rightarrow \text{Vals} \]

Predicates are defined in the standard way:

\[ \sigma, I, L \models \text{pred}(X) \iff \exists \sigma \in (I(\text{pred})(L(X))) \]

Definition 4. \( I \models \Delta \) if \& \( \sigma, I, L \models \Delta \) for all \( \sigma \) and \( L \).

Under mild syntactic restrictions, obeyed in this paper and detailed in [23], one can show that every set of disjoint predicate definitions is satisfiable:

Lemma 5. For any set of definitions \( W_1, \ldots, W_m, D_1, \ldots, D_n \) where \( W_i \) has form \( w_i(\Sigma) = Q_i \) and \( D_j \) is listed in class \( G_j \), there exists an interpretation \( I \) such that \( I = \bigwedge_{i \in 1..m} \forall \Sigma; \ w_i(\Sigma) \iff Q_i \land \bigwedge_{j \in 1..n} \text{apf}_{G_j}(D_j) \) provided that no two distinct definitions in the set define the same predicate.

The semantics of our proof system’s judgements is defined. We do not define the semantics of \( \Gamma_\epsilon \) and \( \Gamma_a \) 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. \( I \models_n \{P\} \sigma \models Q \) if \& \( \sigma, I, L \models \models \) then \( \forall m \leq n \).

1. \( \sigma, \sigma \rightarrow^m \text{fault does not hold, and} \)
2. \( \sigma, \sigma \rightarrow^m \sigma', \epsilon \) then \( \sigma', I, L \models Q \)

The index n deals with mutual recursion in method definitions. \( I \models_n \sigma \) means that all methods in \( \Gamma \) meet their specifications when executed for at least n steps.

Definition 7 (Method verification semantics). If m in G is non-abstrac, let \( \sigma \) denote its body.

\( I, \Gamma \models_\epsilon G.m \rightarrow (\sigma, \{P\}) \sigma(\{Q\}) \) always holds.

\( I, \Gamma \models_{n+1} G.m \rightarrow (\sigma, \{P\}) \sigma(\{Q\}) \) if

\( I \models_n \Gamma \Rightarrow I \models_{n+1} \{P \ast \text{Current} : G\} \sigma(\{Q\}) \)

if G is non-abstrac and true otherwise.

\( I, \Gamma \models_\epsilon G : m \rightarrow (\sigma, \{S\}) \sigma(\{T\}) \) always holds.

\( I, \Gamma \models_{n+1} G : m \rightarrow (\sigma, \{S\}) \sigma(\{T\}) \) if

\( I \models_n \Gamma \Rightarrow I \models_{n+1} \{S\} \sigma(\{T\}) \)
The goal
1. If \( \Delta; \Gamma \vdash_s \{ \text{P} \}\{ \text{Q} \} \) whenever \( \sigma \in \text{spec} \) of \( \text{m} \) we have \( \mathcal{I}, \Gamma \models_n \text{spec} \)
2. If \( \Delta; \Gamma \vdash_s \{ \text{P} \}\{ \text{Q} \} \) then \( \Delta; \Gamma \models \{ \text{P} \}\{ \text{Q} \} \)

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

Lemma 9.
1. If \( \Delta; \Gamma \models \ldots m \ldots \) then \( \forall \mathcal{I} \cdot \text{if} \mathcal{I} \models \Delta \) then for all \( n \) and every spec of \( m \) we have \( \mathcal{I}, \Gamma \models_n \text{spec} \)
2. If \( \Delta; \Gamma \vdash_s \{ \text{P} \}\{ \text{Q} \} \) then \( \Delta; \Gamma \models \{ \text{P} \}\{ \text{Q} \} \)

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

Theorem 11. If a program and its main body \( \sigma \) can be proved with the program verification rule, then \( \forall \mathcal{I}, n \cdot \mathcal{I} \models_n \{ \text{true} \}\{ \text{true} \} \).

Proof.
1. The goal. We have to prove \( \forall \mathcal{I}, n \cdot \mathcal{I} \models_n \{ \text{true} \}\{ \text{true} \} \), which abbreviates \( \forall \mathcal{I}, n \cdot \text{whenever} \sigma, \mathcal{I}, \mathcal{L} \models \text{true} \) then \( \forall m \leq n \cdot 1 \) \( \sigma \) \( \models \text{false} \) fault does not hold, and 2) if \( \sigma, \mathcal{L} \models \text{false} \) \( \sigma', \mathcal{L} \models \text{true} \) this can be simplified to \( \forall n \cdot \sigma, \mathcal{L} \models \text{false} \) fault does not hold.
2. Strengthened assumptions. Let \( \Delta_{T} \triangleq \bigwedge_{i \in [1, n]} \text{apf}(\text{L}_i) \), where \( \text{L}_1 \ldots \text{L}_n \) 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 \( \text{L}_i \), we have \( \Delta_{T}, \Delta_E, \Delta_A; \Gamma \models \text{L}_i \) and \( \Delta_{T} \wedge \Delta_E \wedge \Delta_A; \Gamma \models \{ \text{true} \}\{ \text{true} \} \) also holds for the main body \( \hat{\sigma} \).
3. The interpretation \( \mathcal{T}' \). Since \( \Delta_{T} \wedge \Delta_E \wedge \Delta_A; \Gamma \models \{ \text{true} \}\{ \text{true} \} \), Lemma 9 guarantees \( \Delta_{T} \wedge \Delta_E \wedge \Delta_A; \Gamma \models \{ \text{true} \}\{ \text{true} \} \), which can be simplified to \( \forall \mathcal{I}, n \cdot \text{if} \mathcal{I} \models \Delta_{T} \wedge \Delta_E \wedge \Delta_A \) and \( \mathcal{I} \models_n \Gamma \), then \( \mathcal{I} \models \{ n+1 \cdot \sigma \} \{ \text{true} \} \), which can be instantiated to \( \forall \mathcal{I}, n \cdot \text{if} \mathcal{I} \models \Delta_{T} \wedge \Delta_E \wedge \Delta_A \) and \( \mathcal{I} \models_n \Gamma \), then we can instantiate \( \mathcal{I} \) to \( \mathcal{T}' \) in the formula and simplify to obtain \( \forall n \cdot \sigma, \mathcal{L} \models \text{false} \) fault does not hold. Therefore \( \mathcal{T}' \) serves as a witness that \( \sigma \) will never fault, which is exactly our goal.

Let \( \mathcal{T}' \) be the interpretation whose existence is guaranteed by Lemma 5 for all the where and define clauses in the program. Clearly \( \mathcal{T}' \models \Delta_{T} \). We next prove \( \mathcal{T}' \models \Delta_E \) and then \( \mathcal{T}' \models \Delta_A \).

4. Satisfiability of \( \Delta_A \). Consider an arbitrary export clause \( \mathcal{E} = \text{P where} \{ w_1(x_1) = q_1; \ldots; w_n(x_n) \rightarrow q_n \} \) in class \( \text{L} \). Since \( \text{apf}(\text{L}) \models \mathcal{E} \), we know \( \text{apf}(\text{L}) \wedge \{ \bigwedge_{i \in [1, n]} \forall x_i (x_i \rightarrow q_i) \} \models \mathcal{P} \). The interpretation \( \mathcal{T}' \) satisfies the antecedent, so we also have \( \mathcal{T}' \models \mathcal{P} \). Therefore \( \mathcal{T}' \models \Delta_E \) and \( \mathcal{T}' \models \Delta_{T} \wedge \Delta_E \).

5. Satisfiability of \( \Delta_A \). We prove this by induction. If class \( \text{G} \) has children \( \text{H}_1 \ldots \text{H}_k \), let \( \text{level}(\text{G}) \triangleq 1 + \max(0, \text{level}(\text{H}_1), \ldots, \text{level}(\text{H}_k)) \). Furthermore, \( \text{P}(n) \triangleq \forall \text{G} \) in the program such that \( \text{level}(\text{G}) \leq n \) and for all axiom clauses \( \text{a} \in \text{P} \) in the listing of \( \text{G} \), \( (\Delta_{T} \wedge \Delta_E) \Rightarrow \text{axiominfo}(\text{G}, \text{a} : \text{P}) \).

- Base case. Consider an arbitrary class \( \text{G} \) with \( \text{level}(\text{G}) \leq 1 \) and an axiom clause \( \text{a} \) appearing in it. \( \text{G} \) has no subclasses, and
  (a) If \( \text{G} \) is abstract, there are no objects with dynamic type \( \text{G} \) or a subtype thereof, thus \( \text{axiominfo}(\text{G}, \text{a} : \text{P}) \) holds vacuously and \( \Delta_{T} \wedge \Delta_E \) implies it.
  (b) If \( \text{G} \) is non-abstract, then the only objects whose dynamic type is a subtype of \( \text{G} \) are direct instances of \( \text{G} \). Since \( (\Delta_{T} \wedge \Delta_E \wedge \text{Current} : \text{G}) \Rightarrow \text{P} \) by the Implication premise, we therefore also know \( (\Delta_{T} \wedge \Delta_E) \Rightarrow \text{axiominfo}(\text{G}, \text{a} : \text{P}) \).

Thus \( \text{P}(1) \) holds.

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

So \( \mathcal{T}' \models \Delta_{T} \wedge \Delta_E \wedge \Delta_A \).

6. Wrapping up. We still have to prove \( \forall n \cdot \mathcal{T}' \models_n \Gamma \).

Let \( \mathcal{M} \) be an arbitrary method in the program. Since \( \Delta_{T} \wedge \Delta_E \wedge \Delta_A ; \Gamma \models \mathcal{M} \), by Lemma 9 we know for all \( n \) and every spec of \( m \) that \( \mathcal{T}' \models_n \text{spec} \). Thus \( \forall n \cdot \text{methodspec} \in \mathcal{M} \cdot \mathcal{T}' \models_n \text{methodspec} \), in other words \( \forall n \cdot \mathcal{T}' \models_n \Gamma \).

Acknowledgments

Special thanks to Matthew Parkinson, Peter O’Hearn, Bertrand Meyer, Sebastian Nanz, Carlo Furia and Martin Nordio for feedback on this work. Matthew Parkinson also helped with the implementation of MultiStar’s back-end. Van Staden was
References


