Incremental Reasoning for Multiple Inheritance ⋆

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Department of Informatics, University of Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Object-orientation supports code reuse and incremental programming. Multiple inheritance increases the power of code reuse, but complicates the binding of method calls and thereby program analysis. Behavioral subtyping allows program analysis under an *open world assumption*; i.e., under the assumption that class hierarchies are extensible. However, method redefinition is severely restricted by behavioral subtyping, and multiple inheritance often leads to conflicting restrictions from independently designed superclasses. This paper presents an approach to incremental reasoning for multiple inheritance under an open world assumption. The approach, based on a notion of *lazy behavioral subtyping*, is less restrictive than behavioral subtyping and fits well with multiple inheritance, as it incrementally imposes context-dependent behavioral constraints on new subclasses. We formalize the approach as a calculus, for which we show soundness.

1 Introduction

Object-orientation supports code reuse and incremental programming through inheritance. Class hierarchies are extended over time as subclasses are developed and added. A class may reuse code from its superclasses but it may also specialize and adapt this code by providing additional method definitions, possibly overriding definitions in superclasses. This way, the class hierarchy allows programs to be represented in a compact and succinct way, significantly reducing the need for code duplication. *Late binding* is the underlying mechanism for this incremental programming style; the binding of a method call at run-time depends on the actual class of the called object. Consequently, the code to be executed depends on information which is not statically available. Although late binding is an important feature of object-oriented programming, this loss of control severely complicates reasoning about object-oriented programs.

Behavioral subtyping is the most prominent solution to regain static control of late-bound method calls (see, e.g., [21, 1, 20]), with an *open world assumption*; i.e., where class hierarchies are extensible. This approach achieves incremental reasoning in the sense that a subclass may be analyzed in the context of previously defined classes, such that previously proved properties are ensured by additional verification conditions. However, the approach restricts how methods may be redefined in subclasses. To avoid reverification, any method redefinition must *preserve* certain properties of the method which is redefined. In particular, this applies to the method's contract; i.e., the pre- and

⋆ This research is partially funded by the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl).

$$
\begin{array}{ll}
P ::= \overline{L}\ \{t\} & L ::= \texttt{class}\ C\ \texttt{extends}\ \overline{C}\ \{\overline{f}\ \overline{M}\} \\
M ::= m\ (\overline{x})\{t\} & e ::= \texttt{new}\ C \mid b \mid v \mid \texttt{this} \mid e.m(\overline{e}) \mid m(\overline{e}) \mid m(\overline{e})@C \\
v\ ::= f \mid f@C & t\ ::= v := e \mid \texttt{return}\ e \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{fi} \mid t;t
\end{array}
$$

Fig. 1. The language syntax, with class names C and method names m. Expressions e include fields, **this**, object creation, Boolean expressions b, and method calls. Whitespace is used for list concatenation (i.e., $\overline{e}$ is a list and $e\,\overline{e}$ a non-empty list of expressions).

postcondition for its body. This contract can be seen as a description of the promised behavior of all implementations of the method. Unfortunately, this restriction hinders code reuse and is often violated in practice [30]; for example, it is not respected by the standard Java library definitions.

Multiple inheritance offers greater flexibility than single inheritance, as several class hierarchies can be combined in a subclass. However, it also complicates language design and is often explained in terms of complex run-time data structures such as virtual pointer tables [31], which are hard to understand. Formal treatments are scarce (e.g., [29,8,5,14,32]), but help clarify intricacies, thus facilitating design and reasoning for programs using multiple inheritance. Multiple inheritance also complicates behavioral reasoning, as name conflicts may occur between methods independently defined in different branches of the class hierarchy.

Work on behavioral reasoning about object-oriented programs has mostly focused on languages with single inheritance (see, e.g., [27, 28, 7]). It is an open problem how to design an incremental proof system for multiple inheritance under an open world assumption, without severely restricting code reuse. In this paper we propose a solution to this problem. The approach extends *lazy behavioral subtyping*, which was originally developed for single inheritance systems [13] to allow more flexible code reuse than reasoning systems based on behavioral subtyping. Our approach applies to a wide class of object-oriented systems, relying on the assumption of a *healthy* binding strategy, which is needed for incremental reasoning. Healthiness may easily be imposed on non-healthy binding strategies. The approach is formalized as a syntax-driven inference system, for which we show soundness, combines deductive style program logic with incremental program development, and is well-suited for program development environments. Proofs and a more detailed example may be found in [12].

Paper overview. Sect. 2 introduces late binding and multiple inheritance, Sect. 3 proof environments for behavioral reasoning, and Sect. 4 presents the inference system for incremental reasoning. Sect. 5 discusses methodological aspects, Sect. 6 discusses related work, and Sect. 7 concludes the paper.

2 Late Binding and Multiple Inheritance

An object-oriented kernel language is given in Fig. 1, based on Featherweight Java [16]. For simplicity, we let expressions e be without side-effects and assume that fields f have (locally) distinct names, methods with the same name have the same signature (i.e., no method overloading), class names are unique, programs are well-typed, there is read-

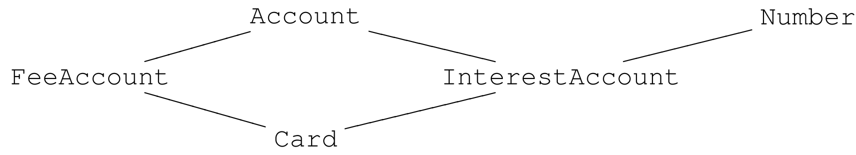

Fig. 2. A multiple inheritance class hierarchy for an account system. The inheritance relation is indicated by lines, e.g., class `FeeAccount` inherits from `Account`.

only access to the formal parameters of methods, as well as **`this`**, and we ignore the types of fields and methods. Two notable differences to Featherweight Java are multiple inheritance and a corresponding form of static method calls. These are explained below. (For brevity, we do not explain standard language features in detail.)

A class C extends a list $\overline{C}$ of superclass names with fields $\overline{f}$ and methods $\overline{M}$, where $\overline{C}$ consists of unique names. We say that C *defines* a method m if the set $\overline{M}$ contains an implementation of m. Let a partial function $body(C,m)$ return this implementation (so $body(C,m)$ is undefined if m is not in $\overline{M}$). For any superclass B of C and method m defined in B, we say that C *inherits* m from B if C does not define m, otherwise m is *overridden* in C. We say that a class C_1 is *below* class C_2 (written $C_1 \leq C_2$) if C_1 and C_2 are the same class or if C_1 extends a class below C_2. Furthermore, C_2 is *above* C_1 if C_1 is below C_2. A *subclass* is below a *superclass*. Two classes are *related* if one is below the other.

There are two kinds of method calls. A *static call* $m(\overline{e})@C$ may occur in a class below C, and it is bound above C at compile time. This statement generalizes the call to the superclass found in languages with single inheritance. In a *remote call* $e.m(\overline{e})$, the object e receives a call to m with actual parameters $\overline{e}$. For convenience, we write $e.m(\overline{e})$ or simply $e.m$ instead of $v := e.m(\overline{e})$ if the result is not needed. Explicit self-calls, written $m(\overline{e})$, are late-bound to **`this`**. Similarly, $f@C$ binds a field f above C.

2.1 Name Conflicts and Healthiness

Inheritance relates classes in a class hierarchy. For single inheritance this hierarchy forms a tree, whereas for multiple inheritance, the hierarchy forms a directed, acyclic graph. In the single inheritance tree, *vertical* name conflicts occur when a subclass overrides a method from a superclass. The *binding strategy* for method calls must resolve such conflicts. Late binding or dynamic dispatch selects the method body to be executed at run-time, depending on the callee's run-time class: the selected body is found by the *first matching definition* of the method above the actual class of the object. In multiple inheritance class hierarchies there are also *horizontal* name conflicts. These occur when different definitions of the same method are found above a given class, depending on the chosen path through the hierarchy. More elaborate binding strategies are needed to resolve horizontal conflicts. Some binding strategies are infeasible, as they contradict incremental program development. This is illustrated by the following example.

Example 1. We consider a class hierarchy for a bank account system, given in Fig. 2. Potential problems with horizontal name conflicts are illustrated by the classes in Fig. 3,

```
class Account { int bal = 0;
  deposit(int x) {...;update(x)}
  withdraw(int x) {...;update(-x)}
  update(int y) {...;bal=bal+y;...}}

class Number { int num;
  update(int x) {num = x }
  increase(int x) {update(num+x)}}

class InterestAccount extends Account Number { int fee;
  addInterest(int x y) {...; deposit(x);increase(y)}
  withdraw(int x) {withdraw(x)@Account;if bal<0 then update(-fee) fi}}

class FeeAccount extends Account { int fee;
  withdraw(int x) {withdraw(x)@Account;update(-fee) }
  update(int y) {...; bal=bal+y;...}}

class Card extends FeeAccount InterestAccount {
  withdraw(int x) {withdraw(x)@InterestAccount;update(-fee@FeeAccount)}}
```

Fig. 3. Implementation sketches of the classes in Fig. 2.

sketching an implementation of the account system. (A more detailed implementation is available in the extended version of this paper [12].) Class `Account` implements basic facilities for depositing and withdrawing money. The actual manipulation of the balance is implemented by a method `update`. Class `Number`, developed independently of `Account`, provides functionality for manipulating a number `num`. These two classes are inherited by the subclass `InterestAccount`, where field `num` plays the role of the current interest rate. Method `addInterest` increases the interest rate after depositing a value. For `InterestAccount`, inheritance of `Account` and `Number` gives a horizontal name conflict for method `update`. The behavior of the two versions of `update` is completely different, which means that the behavior specified by `increase` in `Number` will not hold in the subclass, if the self-call to `update` in `increase` is bound to `Account`. Thus, in order to support incremental design, the self-call in `Number` should bind to the definition in `Number`, and correspondingly for the self-call in `Account`should bind to the definition in `Account`.

One solution to resolve horizontal name conflicts is *explicit resolution* specified as part of an inheritance list; e.g., to use qualification or renaming as in C++ [31], Eiffel [23], and POOL [2]. However, it might be undesirable to force the programmer to modify method names, making programs more difficult to understand and maintain. We generalize this approach and decorate each self-call with a *binding clause* restricting the binding space. Such a clause may represent a specific name resolution strategy, or be explicitly provided by the programmer. This way, the approach of this paper is applicable to several resolution strategies. Binding clauses allow us to consider horizontal name conflicts as a natural feature of multiple inheritance. In particular when using libraries, the programmer cannot be expected to know (or resolve) potential name conflicts of, e.g., auxiliary methods in the libraries. To support incremental program development and reasoning, we impose the following *healthiness condition* on the binding strategy:

– a self-call made by a method defined in C must bind to a class related to C, and

– a remote call $x.m$, where x has C as declared class, must bind to a class related to C.

It is easy to see that healthiness removes accidental overriding of methods, due to unfortunate binding. Let $C\#m$ denote a call to m where the binding is restricted to classes related to C. In Example 1, if the call to `update` in `Number` is replaced by `Number#update`, the call becomes healthy. When executed in an instance of `InterestAccount`, the call will bind related to `Number`. For the rest of the paper, we use the convention that a self-call to m made by a method defined in C is understood as $C\#m$. Similarly, a remote call $x.m$ with C as the declared class of x, is understood as $x.C\#m$. As static calls are inherently healthy, this ensures healthy binding. A particular binding strategy is given below. We here assume that the notation $C\#m$ and $x.C\#m$ is introduced during static analysis, but it could also be made available to the programmer.

2.2 The Binding of Method Calls and Fields

For the reasoning system, we need an explicit definition of a healthy resolution strategy. In this paper, we formalize the strategy by a function *bind* defined below. Other definitions of *bind* are possible and would lead to variations of the calculus. A call to a method m is bound with respect to a *search class* D; i.e., $bind(D,m)$, where the search for a definition of m starts in D. Following [9, 11, 17], ambiguities are solved by fixing the order in which inherited classes are searched, e.g., from left to right. Let *Cid* and *Mid* denote class and method names. To make the representation of class hierarchies compact, a class name is bound to a tuple $\langle \overline{C}, \overline{f}, \overline{M} \rangle$ of type *Class*, where $\overline{C}$, $\overline{f}$, and $\overline{M}$ are accessible by the observer functions *inh*, *fields*, and *mtds*, respectively. This binding strategy can be defined by a partial function *bind*: $List[Cid] \times Mid \rightarrow Cid$:

$$\begin{array}{lll} bind(nil,m) & \triangleq \bot & \\ bind(D\,\overline{D},m) & \triangleq D & \text{if } m \in D.mtds \\ bind(D\,\overline{D},m) & \triangleq bind(D.inh\,\overline{D},m) & \text{otherwise,} \end{array}$$

where *nil* denotes the empty list and $D.inh\,\overline{D}$ reduces to $\overline{D}$ when $D.inh$ is empty. This strategy is not healthy, since a self-call would be bound independently of where in the hierarchy the call is made. A healthy strategy can be obtained by restricting the binding to classes related to the class where the call is made. We let the notation $bind(D,C\#m)$ define the call $C\#m$ for search class D. The search is restricted by C; the returned class must be either above or below C. This ensures the healthiness condition described above. By type-safety, there is a definition of m above C; thus $bind(D,C\#m)$ is well-defined for D below C.

Definition 1. *Define* $bind(_,_\#_) : List[Cid] \times Cid \times Mid \rightarrow Cid$ *by:*

$$\begin{array}{lll} bind(nil,C\#m) & \triangleq \bot & \\ bind(D\overline{D},C\#m) & \triangleq D & \text{if } (D < C \vee D \geq C) \wedge m \in D.mtds \\ bind(D\overline{D},C\#m) & \triangleq bind(D.inh\overline{D},C\#m) & \text{if } (D < C \vee D \geq C) \wedge m \notin D.mtds \\ bind(D\overline{D},C\#m) & \triangleq bind(\overline{D},C\#m) & \text{otherwise} \end{array}$$

A *remote call* $x.m$ is bound by $bind(D, C\#m)$ where C is the declared class of x and D the actual class of x. A *statically bound* method call $m@C$ is bound above C independently of the actual class that the call is executed in. Following the traversal strategy above, the binding of the call is given by $bind(m@C)$:

Definition 2. *Define* $bind(_@_) : Mid \times Cid \to Cid$ *by:* $bind(m@C) \triangleq bind(C, C\#m)$.

Similar binding functions may be used to define the binding of fields: An occurrence of $f@B$ is allowed inside a class declaration C if B is above C, and is bound above B; and an unqualified occurrence of f inside C is understood as $f@C$.

3 Lazy Behavioral Subtyping

Lazy behavioral subtyping supports incremental reasoning for extensible class hierarchies; each class is analyzed based on the analysis of its superclasses, but independent of (future) subclasses. Lazy behavioral subtyping was presented for single inheritance in [13]. We here present an extension for multiple inheritance and horizontal name conflicts, assuming a healthy binding strategy. With healthy binding, a method call binds to a class related to the calling class. Therefore behavioral constraints may be propagated down the class hierarchy, which allows incremental reasoning. The proof method has two parts, a conventional program logic (e.g., [27, 15, 4, 25]) and, on top of that, a *proof environment* which incrementally tracks method specifications and requirements.

The proof system uses Hoare triples $\{p\}\,t\,\{q\}$ with the standard partial correctness interpretation: if a statement t starts execution in a state where a precondition p holds and this execution terminates, then the postcondition q holds afterwards. Triples can be derived in any suited program logic, so let $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$ denote that the triple $\{p\}\,t\,\{q\}$ is derivable in the chosen program logic PL. A *proof outline* [25] for a method definition $m(\bar{x})\{t\}$ is an annotated method definition $m(\bar{x}) : (p,q)\,\{t\}$, where method calls inside t are decorated with call-site requirements. We henceforth assume that all method bodies are decorated in this way. The derivability $\vdash_{\mathrm{PL}} m(\bar{x}) : (p,q)\,\{t\}$ of a proof outline is given by $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$. Let *Spec* denote pairs (p,q) of conditions.

Method specifications and requirements. The verification technique distinguishes between a method's declared specification (its contract) and its *requirement*. Roughly, the first captures its announced behavior as declared in the pre- and post-condition of the method definition. In contrast, the requirements stem from call-sites and represent properties needed to verify the client code of a method, namely to satisfy the client code's specification in turn. Due to inheritance and overriding, a method with a given name is available in more than one class, and can be called from different client codes. Consequently, the properties are considered per class and its position in the class hierarchy. If, furthermore, the class hierarchy is incrementally extended, new specifications and requirements may be added. This bookkeeping of the properties is done in a proof environment, through the two mappings S and R.

Definition 3 (Proof environments). *A* proof environment *is a triple* $\langle P, S, R\rangle$ *of type Env, where* $P : Cid \to Class$ *is a partial mapping and* R *and* S *are total mappings of type* $Cid \times Cid \times Mid \to Set[Spec]$.

In such a proof environment $\mathcal{E}$, the mapping P reflects the class hierarchy and the two mappings S and R contain the constraints collected so far during analysis. We use a subscript, e.g., $R_{\mathcal{E}}$, if the proof environment is not clear from the context.

For a method m defined in a class B, besides m's declared specification as given in B itself, subclasses of B may give *additional* specifications for the method. For example, if a method n is overridden by a subclass C of B, and m calls n, a specification of $body(B,m)$ given by C may account for m's behavior relying on the overriding version of n. Hence, for a method m defined in B, $S(C,B.m)$ represents the specification as given in C. Note that a non-empty $S(C,B.m)$ implies $C \leq B$.

Example 2. Recall the method `update`, implemented in both `Account` and `Number` in Example 1. Let the specifications of these two definitions of `update` be contained in $S($`Account`,`Account.update`$)$ and $S($`Number`,`Number.update`$)$, respectively. The common subclass `InterestAccount` may provide *additional* specifications for these implementations in the sets $S($`InterestAccount`,`Account.update`$)$ and $S($`InterestAccount`,`Number.update`$)$.

In order to preserve a declared specification $(p,q) \in S(C,B.m)$ when inheriting m, it is necessary to impose requirements on methods called via late binding in $body(B,m)$. The requirements are given by the proof outline $m(\bar{x}) : (p,q)\, \{body(B,m)\}$ and maintained by the requirement mapping R. For each call $\{r\}\, n()\, \{s\}$ in this outline, the requirement (r,s) is included in $R(C,B\#n)$. Here, C denotes the class that *imposes* the requirement and B is the call-site class where m is defined.

Since we work with sets of specifications, the entailment relation is lifted as follows. Let p' be the condition p with all fields f substituted by f', avoiding name capture.

Definition 4 (Entailment). *Assume specifications (p,q) and (r,s), and specification sets $\mathcal{U} = \{(p_i,q_i) \mid 1 \leq i \leq n\}$ and $\mathcal{V} = \{(r_i,s_i) \mid 1 \leq i \leq m\}$.* Entailment *is defined by*

i) $(p,q) \rightarrow (r,s) \triangleq (\forall \bar{z}_1 \,.\, p \Rightarrow q') \Rightarrow (\forall \bar{z}_2 \,.\, r \Rightarrow s')$,
where $\bar{z}_1$ and $\bar{z}_2$ are the logical variables in (p,q) and (r,s), respectively

ii) $\mathcal{U} \rightarrow (r,s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall \bar{z}_i \,.\, p_i \Rightarrow q_i')) \Rightarrow (\forall z \,.\, r \Rightarrow s')$.

iii) $\mathcal{U} \rightarrow \mathcal{V} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrow (r_i,s_i)$.

The relation $\mathcal{U} \rightarrow (r,s)$ corresponds to Hoare-style reasoning, proving $\{r\}\, t\, \{s\}$ from $\{p_i\}\, t\, \{q_i\}$ for all $1 \leq i \leq n$, by means of the adaptation and conjunction rules [3]. Entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \rightarrow \mathcal{V}$.

Soundness. It is crucial for incremental reasoning to preserve the declared specifications for *inherited* methods: for a specification (p,q) included in $S(C,B.m)$ it is safe to rely on (p,q) when $body(B,m)$ is executed on an instance of subclasses of C. Note that *overriding* implementations of m in such subclasses may satisfy different contracts than the definition in the superclass. This flexibility goes beyond standard behavioral subtyping. With the open world assumption the subclasses of C are unknown when C is analyzed, so soundness is ensured by tracking the requirements that (p,q) imposes on late-bound calls in $body(B,m)$. If n is overridden in a class D below C, all requirements towards n made by classes above D must be satisfied by $body(D,n)$. This is expressed

by $S(D, D.n) \twoheadrightarrow R{\uparrow}(D,n)$, where $R{\uparrow}(D,n)$ denotes the union of all requirements towards n made above D; i.e., the union of $R(C, B\#n)$ for all $D \leq C \leq B$.

In general soundness means that if $body(B,m)$ is executed on an instance of class D, it must be safe to rely on $S{\uparrow}(D, B.m)$, which is the union of $S(C, B.m)$ for all classes C where $D \leq C \leq B$. Soundness is formalized by the following definition of sound proof environments and Lemma 1. Let $C \in \mathcal{E}$ denote that $P_{\mathcal{E}}(C)$ is defined, and $x : C.m$ the remote call $x.m$ where x is declared with type C.

Definition 5 (Sound environments). *Let $B,C,D : Cid$ and $m,n : Mid$. A* sound environment *$\mathcal{E}$ satisfies the following two conditions for all $B,C \in \mathcal{E}$ and m:*

$$\begin{array}{l} i)\ \forall (p,q) \in S_{\mathcal{E}}(C, B.m)\ .\ \exists body(B,m)\ .\ \vdash_{\mathrm{PL}} m(\bar{x}) : (p,q)\,\{body(B,m)\} \\ \quad \wedge\ Local_{\mathcal{E}}(C,B,body(B,m)) \wedge Rem_{\mathcal{E}}(body(B,m)) \wedge Stat_{\mathcal{E}}(C, body(B,m)) \\ ii)\ m \in C.\mathrm{mtds} \Rightarrow S_{\mathcal{E}}(C, C.m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(C,m) \end{array}$$

where

$$\begin{array}{l} Local_{\mathcal{E}}(C,B,t) \triangleq \forall \{r\}\, n\, \{s\} \in t\ .\ \forall D \leq_{\mathcal{E}} C\ .\ S{\uparrow}_{\mathcal{E}}(D, bind(D, B\#n).n) \twoheadrightarrow (r,s) \\ Rem_{\mathcal{E}}(t) \triangleq \forall \{r\}\, x : D.n\, \{s\} \in t\ .\ S{\uparrow}_{\mathcal{E}}(D, bind(n@D).n) \twoheadrightarrow (r,s) \wedge\ R{\uparrow}_{\mathcal{E}}(D,n) \twoheadrightarrow (r,s) \\ Stat_{\mathcal{E}}(C,t) \triangleq \forall \{r\}\, n@B\, \{s\} \in t\ .\ S{\uparrow}_{\mathcal{E}}(C, bind(n@B).n) \twoheadrightarrow (r,s) \end{array}$$

The soundness of a proof environment can be explained informally as follows: Assume that $(p,q) \in S_{\mathcal{E}}(C, B.m)$ and that there is a proof outline of $body(B,m)$ for (p,q). For each requirement $\{r\}\, n\, \{s\}$ to a *self-call* in this proof outline and for each subclass D of C, (r,s) must follow from the specifications of the method definition to which a call is bound for search class D. For each requirement $\{r\}\, x.n\, \{s\}$ to a *remote call*, (r,s) must follow from the specification of the method provided by the static type of x, and it must be imposed on redefinitions below the static type. For each requirement $\{r\}\, n@A\, \{s\}$ to a *static call*, (r,s) must follow from the specification of the method implementation to which the call will bind. The requirement is not imposed on method overridings since the call is bound at compile time.

Let $\models_C \{p\}\, t\, \{q\}$ denote $\models \{p\}\, t\, \{q\}$ provided that late-bound self-calls in t are bound for search class C, and let $\models_C m(\bar{x}) : (p,q)\, \{t\}$ be given by $\models_C \{p\}\, t\, \{q\}$. If t is without calls and $\vdash_{\mathrm{PL}} \{p\}\, t\, \{q\}$, then $\models \{p\}\, t\, \{q\}$ follows by the soundness of PL. Lemma 1 states that if $(p,q) \in S_{\mathcal{E}}(C, B.m)$ and $body(B,m)$ is executed in an instance of a subclass D of C, a sound environment guarantees that (p,q) is a valid specification:

Lemma 1. *Assume given a sound environment $\mathcal{E}$ and a sound program logic* PL. *Let $B,D : Cid$, $m : Mid$, and $(p,q) : Spec$ such that $B,D \in \mathcal{E}$ and $(p,q) \in S{\uparrow}_{\mathcal{E}}(D, B.m)$. Then $\models_D m(\bar{x}) : (p,q)\, \{body_{\mathcal{E}}(B,m)\}$.*

Example 3. Consider the method `Account.withdraw`(x), specified by $(\texttt{bal} = b_0, \texttt{bal} = b_0 - x) \in S(\texttt{Account}, \texttt{Account.withdraw})$. This specification leads to a requirement on `update`: the method modifies the balance according to its parameter. The requirement is satisfied by `update` defined in `Account`, and `FeeAccount`, the two implementations to which the call in `Account` can be bound. The separation of method specifications from requirements made by method calls allows incremental reasoning without imposing the constraints of behavioral subtyping on method

overridings. For instance in `FeeAccount`, the overriding implementation satisfies ($\mathtt{bal} = b_0, \mathtt{bal} = b_0 - x - \mathtt{fee@FeeAccount}$). Incremental reasoning is still supported as the static call to the superclass method relies on the verified specification of `withdraw` in `Account`. Correspondingly for the implementations of `withdraw` in `InterestAccount` and `Card`.

4 The Inference System for Incremental Reasoning

The inference system analyzes and manipulates the proof environments. Establishing a proof outline for one method at a given stage of the overall analysis gives rise to (further) proof-obligations, which are tracked by the proof system (cf. Section 4.1). The system itself is formalized as a set of derivation rules (cf. Section 4.3), whose traversal through the class-hierarchy is driven by the analysis operations given in Section 4.2.

4.1 Tracking behavioral constraints

Assume that a proof outline $m(\overline{x}) : (p,q)\, \{body(B,m)\}$ is given by a class C. To ensure soundness, this gives rise to the following steps:

1. (p,q) is included in $S(C, B.m)$.
2. for each call $\{r\}\, n\, \{s\}$ in the proof outline:
   (a) (r,s) is analyzed with regard to the implementation of $B\#n$ found for search class C; i.e., the proof obligation $S{\uparrow}(C, E.n) \rightharpoonup (r,s)$ must be established, where $E = bind(C, B\#n)$.
   (b) (r,s) is included in $R(C, B\#n)$.

Establishing $S{\uparrow}(C, E.n) \rightharpoonup (r,s)$ in step 2a means: Either (r,s) follows directly from the already established specifications in $S{\uparrow}(C, E.n)$ by entailment, or the proof outline $n(\overline{y}) : (r,s)\, \{body(E,n)\}$ given by C is analyzed in the same manner as the original specification of m. This adds (r,s) to $S(C, E.n)$, trivializing the proof of $S{\uparrow}(C, E.n) \rightharpoonup (r,s)$.

Including (r,s) into $R(C, B\#n)$ in step 2b constrains future subclasses of C: Each subclass D of C must ensure

$$S{\uparrow}(D, bind(D, B\#n).n) \rightharpoonup (r,s) \tag{1}$$

If n is overridden by D, all late-bound calls to n made by classes above D will bind to the definition of n in D. As explained, the calculus then ensures (1) by establishing $S(D, D.n) \rightharpoonup R{\uparrow}(D, n)$. If n is not overridden by D, we distinguish two cases. Let $E = bind(D, B\#n)$; i.e., the call $B\#n$ will bind to the implementation in E for search class D. If E is *related* to C, soundness of the analysis of superclasses of D ensures (1). If otherwise E is *unrelated* to C, class D may introduce a *diamond* in the class hierarchy, which needs to dealt with. A diamond is introduced by D if there are two different classes D_1 and D_2 in $D.inh$ and a class A such that $D_1 \leq A$ and $D_2 \leq A$. Let $commSup(D)$ denote the union of all such classes A. For an E unrelated to C, let $B \in commSup(D)$. Then the requirements $R(C, B\#n)$ were not imposed on $body(E,n)$ at the time E was analyzed. For soundness, they are therefore imposed on $body(E,n)$ when the diamond is

created by D. More generally, the same argument applies to all classes between D and B that are unrelated to E. We let the set $dreq(D,B\#n)$ of *diamond requirements* denote the union of all $R(C,B\#n)$ for C such that $D \leq C \leq B$ and $bind(D,B\#n)$ is unrelated to C. By the analysis of D, the calculus ensures (1) by establishing $S\!\uparrow\!(D,E.n) \multimap dreq(D,B\#n)$. Note that in the subcase where E is related to C, class D may also introduce a diamond. This case is covered by the proof of Theorem 1 (details are in the extended version [12]).

Example 4. In the classes of Example 1, the method `update` is defined in `Account` and overridden in `FeeAccount`. Let class `InterestAccount` impose a requirement (r,s) on `update`, contained in $R(\texttt{InterestAccount},\texttt{Account\#update})$. Now the class `Card` introduces a diamond in the class hierarchy. Since class `Card` is a subclass of `InterestAccount`, soundness requires the validity of the formula $S\!\uparrow\!(\texttt{Card},bind(\texttt{Card},\texttt{Account\#update}).\texttt{update}) \multimap (r,s)$ where the method binding resolves to `FeeAccount`. Since `FeeAccount` and `InterestAccount` are unrelated, (r,s) is in the set $dreq(\texttt{Card},\texttt{Account\#update})$, and the calculus establishes the required verification of (r,s) by the analysis of `Card`.

4.2 Analysis Operations

The judgments of the calculus are of the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a list of *analysis operations* with the following syntax.

$$\begin{aligned}
O &::= \varepsilon \mid anMtd(\overline{M}) \mid anOutln(C,t) \mid verify(C,m,\overline{R}) \mid supCls(\overline{C}) \mid supMtd(C,\overline{m}) \mid O \cdot O \\
\mathcal{S} &::= \emptyset \mid L \mid require(C,m,(p,q)) \mid \mathcal{S} \cup \mathcal{S} \\
\mathcal{A} &::= module(\overline{L}) \mid [\langle C : O\rangle \,;\, \mathcal{S}] \mid [\varepsilon \,;\, \mathcal{S}] \mid module(\overline{L}) \cdot \mathcal{A}
\end{aligned}$$

Here L denotes a class definition, as defined in Fig. 1. The rule system below specifies an algorithm that traverses a class hierarchy and its syntactic constituents — classes, methods, statements, etc. — according to the principles explained above; in particular, tracking specifications and requirements. The analysis starts with an $\mathcal{E} \vdash \mathcal{A}$ where $\mathcal{E}$ is empty and $\mathcal{A}$ contains the program as a sequence of modules. A module is a set of classes considered as a *compilation unit*. At each stage of the development, the modules given so far represent a complete, compilable program. Programs are open in the sense that new modules may be analyzed at later stages. Inside a module, the set $\mathcal{S}$ contains a module's classes. The inference rules ensure that a class can only be analyzed after analysis of all its superclasses.

The above operations and the proof environment drive the algorithm through the program. The operation **`class`** C **`extends`** $\overline{D}$ $\{\bar{f}\ \overline{M}\}$ initiates the analysis of C, and $[\langle C : O\rangle \,;\, \mathcal{S}]$ analyzes O in the context of class C *before* operations in $\mathcal{S}$ are considered. The analysis of a specific class involves the analysis of the proof outlines for its methods $\overline{M}$, the verification of the requirements for a method, and collecting the proof obligations for the calls mentioned inside the method bodies (by the operations $anMtd(\overline{M})$, $verify(D,m,\overline{R})$, and $anOutln(D,t)$). The operation $require(D,m,(p,q))$ applies to remote calls to ensure that m in D satisfies the requirement (p,q). Requirements are lifted outside the context of the analyzed class by this operation, and shifted into the set $\mathcal{S}$ of analysis operations. The two remaining operations, $supCls(\overline{D})$ and $supMtd(D,\overline{m})$ are only used during analysis of C, if C introduces diamonds.

Environment updates. Updates are represented by the operator $_ \oplus _ : Env \times Update \rightarrow Env$, where the second argument represents the update. There are three different environment updates; loading a new class and extending the specifications or the requirements of a method in a class. The updates are defined as follows:

$$\begin{aligned}
&\mathcal{E} \oplus \mathbf{extP}(C,\overline{D},\overline{f},\overline{M}) = \langle P_{\mathcal{E}}[C \mapsto \langle \overline{D},\overline{f},\overline{M}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
&\mathcal{E} \oplus \mathbf{extS}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}[(C,D,m) \mapsto S_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}], R_{\mathcal{E}}\rangle \\
&\mathcal{E} \oplus \mathbf{extR}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C,D,m) \mapsto R_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}]\rangle
\end{aligned}$$

4.3 The Inference Rules

The inference rules are given in Fig. 4. Rule (NEWMODULE) initiates the analysis of a set of classes. For convenience, we let $\overline{L}$ denote both a list and set of classes. Furthermore, (NEWCLASS) loads a new class C for analysis, the second premise ensures that the superclasses $\overline{D}$ have already been analyzed. For each method m in C, the calculus generates an operation $verify(C,m,\overline{R})$, where $\overline{R}$ is the set of requirements that must hold for this method. Rules (REQDER) and (REQNOTDER) deal with the verification of a particular specification with respect to the implementation. If the specification follows from the already established specification of the method, rule (REQDER) continues with the remaining analysis operations. Otherwise, a proof of the specification is required. By (REQNOTDER), an outline of the specification is then analyzed by an *anOutln* operation. Remark that only rule (REQNOTDER) extends the S mapping.

For a given proof outline, the rules (LATECALL), (STATCALL), and (REMCALL) handle late-bound, static, and remote calls, respectively. Rule (LATECALL) extends the R mapping and generates a *verify* operation to analyze the requirement for the implementation to which the call will bind. The extension of R ensures that the requirement will be respected by future subclasses. Rule (STATCALL) also generates a *verify* operation, but does not extend R. Remote late-bound calls are handled by the rules (REMREQ) and (REMCALL), which allow reasoning from the method requirements given in the declared class of the callee. Notice that no new requirements are imposed. However, as requirements are generated from internal self-calls in a class, these may not provide suitable external specifications.

Finally, there are rules for analyzing requirements from common superclasses when diamonds are introduced in the environment. Rule (SUPMTD) generates a *supMtd* for each common superclass. For each method called by a common superclass, (SUPREQ) generates a *verify* operation for the requirements imposed by calls to the method. If a class introduced by (NEWCLASS) does not have any common superclasses, the generated *supCls* operation will have an empty argument and can be discarded by (NOSUP).

For brevity, we elide a few straightforward rules which formalize a lifting from single-elements to sets or sequences of elements. For example, the rule for $anMtd(\overline{M})$ (which occurs in the premise of (NEWCLASS)), generalizes the analysis of a single method which is done in (NEWMTD). These rules are included in the extended version of this paper [12], together with the proof of the soundness theorem below. Note that a proof of $\mathcal{E} \vdash module(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\varepsilon\,;\emptyset]$; we call $\mathcal{E}'$ the environment resulting from the analysis of $module(\overline{L})$.

Theorem 1. *Let $\mathcal{E}$ be a sound environment and $\overline{L}$ a set of class declarations. If a proof of $\mathcal{E} \vdash module(\overline{L})$ has $\mathcal{E}'$ as its resulting environment, then $\mathcal{E}'$ is also sound.*

$$\frac{C \notin \mathcal{E} \qquad \overline{D} \neq \text{nil} \Rightarrow \overline{D} \in \mathcal{E} \qquad \overline{E} = \mathit{commSup}_{\mathcal{E}}(C) \qquad \mathcal{E} \oplus \mathbf{extP}(C, \overline{D}, \overline{f}, \overline{M}) \vdash [\langle C : \mathit{anMtd}(\overline{M}) \cdot \mathit{supCls}(\overline{E}) \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon \,;\, \{\texttt{class}\ C\ \texttt{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\}\} \cup \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{NewClass})$$

$$\frac{\mathcal{E} \vdash [\varepsilon \,;\, \overline{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash \mathit{module}(\overline{L}) \cdot \mathcal{A}}\ (\textsc{NewModule})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(C, m, \{(p,q)\} \cup R{\uparrow_{\mathcal{E}}}(C.\mathit{inh}, m)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{anMtd}(m(\overline{x}) : (p,q)\, \{t\}) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{NewMtd})$$

$$\frac{\mathcal{E} \vdash [\varepsilon \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \varepsilon \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{EmpClass})$$

$$\frac{S{\uparrow_{\mathcal{E}}}(C, D.m) \twoheadrightarrow (p,q) \qquad \mathcal{E} \vdash [\langle C : O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, (p,q)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{ReqDer})$$

$$\frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\varepsilon \,;\, \emptyset] \cdot \mathcal{A}}\ (\textsc{EmpModule})$$

$$\frac{\vdash_{\mathrm{PL}} m : (p,q)\, \{\mathit{body}_{\mathcal{E}}(D,m)\} \qquad \mathcal{E} \oplus \mathbf{extS}(C, D, m, (p,q)) \vdash [\langle C : \mathit{anOutln}(D, \mathit{body}_{\mathcal{E}}(D,m)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, (p,q)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{ReqNotDer})$$

$$\frac{E = \mathit{bind}(C, D\#m) \qquad \mathcal{E} \oplus \mathbf{extR}(C, D, m, (p,q)) \vdash [\langle C : \mathit{verify}(E, m, (p,q)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{anOutln}(D, \{p\}\, m\, \{q\}) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{LateCall})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(\mathit{bind}(m@B), m, (p,q)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{anOutln}(D, \{p\}\, m@B\, \{q\}) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{StatCall})$$

$$\frac{\mathcal{E} \vdash [\langle C : O \rangle \,;\, \mathcal{S} \cup \{\mathit{require}(E, m, (p,q))\}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{anOutln}(D, \{p\}\, x : E.m\, \{q\}) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{remCall})$$

$$\frac{C \in \mathcal{E} \qquad R{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow (p,q) \qquad S{\uparrow_{\mathcal{E}}}(C, \mathit{bind}(m@C).m) \twoheadrightarrow (p,q) \qquad \mathcal{E} \vdash [\varepsilon \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon \,;\, \{\mathit{require}(C, m, (p,q))\} \cup \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{remReq})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \mathit{called}_{\mathcal{E}}(D) \setminus C.\mathit{mtds}) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(D) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{SupMtd})$$

$$\frac{E = \mathit{bind}(C, D\#m) \qquad \mathcal{E} \vdash [\langle C : \mathit{verify}(E, m, \mathit{dreq}(C, D\#m)) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, m) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{SupReq})$$

$$\frac{\mathcal{E} \vdash [\langle C : O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\emptyset) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{NoSup}) \qquad \frac{\mathcal{E} \vdash [\langle C : O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \emptyset) \cdot O \rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}\ (\textsc{NoSupMtd})$$

Fig. 4. The inference system. Here m denotes a call, including actual parameters, and $\mathit{called}(D)$ denotes the names of the methods called by D.

5 Methodological Aspects

With the given approach, a programmer typically provides S-requirements for each class. Their verification generates R-requirements for the late-bound self-calls occurring in the class, which will be imposed on subclass redefinitions of the called methods. In a subclass *C*, redefined methods can violate the S-requirements of a superclass, but not the R-requirements. *C* may give additional contracts for inherited methods, resulting in additional verification of such methods, which may generate additional R-requirement for future subclasses of *C*. With multiple inheritance, this means that different parts of the inheritance graph may have different R-requirements to the same method. Note that behavioral subtyping is not implied by this approach: When *m* is overridden, the new definition need not implement all superclass specifications of *m*, but only the R-requirements made towards usage of *m*. This way, lazy behavioral subtyping still supports incremental reasoning under an open world assumption.

A weakness of the approach presented here is that a remote call $x.m(..)$ may create R-requirements to *m* for the declared class of *x*, say *C*, and these requirements must be imposed on *C* and its subclasses, unless they follow from already established R-requirements to *m* for *C*. Adding R-requirements to a previously established class hierarchy can lead to several verification tasks, which makes the approach less modular. As R-requirements generated from internal self-calls in a class may not in general provide suitable external properties, a programmer should provide R-requirements such that reasoning about remote calls can be derived from these. Therefore a programmer should be aware of the distinction between S- and R-requirements, and be able to provide both, and that unnecessarily strong R-requirements will restrict future method redefinitions.

A more modular version of lazy behavioral subtyping may be obtained by using *behavioral interfaces*. A behavioral interface describes the visible methods of a class and their contracts (or possibly an invariant), and inheritance may be used to form new interfaces from old ones. An advantage of seeing all classes through interfaces is that explicit hiding constructs become superfluous. A class may then be specified by a number of interfaces. If all object variables (references) are typed by interfaces, one may let the inheritance hierarchies of interfaces and classes be independent. In particular, one need not require that a subclass of *C* inherits (nor respects) the behavioral interfaces specified for *C*: Static type checking of an assignment $x := e$ must then ensure that the expression *e* denotes an object supporting the declared interface of the object variable *x*. In this setting, the substitution principle for objects can be reformulated as follows: *For an object variable x with declared interface I, the actual object referred to by x at run-time will satisfy the behavioral specification I.* As a consequence, a subclass may freely reuse and redefine superclass methods, since it is free to violate the behavioral specification of superclasses. Reasoning about a remote call $x.m(..)$ can then be done by relying on the behavioral interface of the object variable *x*, simplifying rule (REMREQ) to simply check interface contracts. This approach is followed by, e.g., Creol [18].

6 Related Work

Multiple inheritance is supported in, e.g., C++ [31], CLOS [11], Eiffel [23], POOL [2], and Self [9]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by ex-

plicit resolution, after which the inheritance graph may be linearized. Multiple dispatch, or multi-methods [11], gives a more powerful binding mechanism, but reasoning about multi-methods and redefinition is difficult. The prototype-based language Self [9] proposes an elegant *prioritized binding strategy*. Each superclass is given a priority. With equal priority, the superclass related to the caller class is preferred. However, explicit class priorities may cause surprises in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller, binding fails.

Formalizations of multiple inheritance in the literature traditionally use the *objects-as-records* paradigm. This approach addresses subtyping issues related to subclassing, but method binding is not easily captured. In Cardelli's denotational semantics of multiple inheritance [8], not even access to methods of superclasses is addressed. Rossie, Friedman, and Wand [29] formalize multiple inheritance using *subobjects*, a run-time data structure used for virtual pointer tables [19,31]. This work focuses on compile-time issues and does not clarify multiple inheritance at the abstraction level of the programming language. A natural semantics for late binding in Eiffel models the binding mechanism at the abstraction level of the program [5]. Recently, an operational semantics and type safety proof inspired by C++ has been formalized in Isabelle [32].

Work on behavioral reasoning about object-oriented programs address languages with single inheritance (e.g., [27,28,7]). For late binding, different variations of behavioral subtyping are most common [21,1,20], as discussed above. Pierik and de Boer [27] present a sound and complete reasoning system for late-bound calls which does not rely on behavioral subtyping. This work, also for single inheritance, is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental reasoning.

Lately, incremental reasoning, both for single and multiple inheritance, has been considered in the setting of *separation logic* [22, 10, 26]. These approaches support a distinction between static specifications, given for each method implementation, and dynamic specifications that are used to verify late-bound calls. The dynamic specifications are given at the declaration site, in contrast to our work where late-bound calls are verified based on call-site requirements.

7 Conclusion and Future Work

Lazy behavioral subtyping supports incremental reasoning under an open world assumption, where class hierarchies can be gradually extended by inheritance. The approach is more flexible than traditional behavioral subtyping, as illustrated by the running example. This paper has introduced a healthiness condition for method binding and extended lazy behavioral subtyping to the setting of multiple inheritance for healthy binding strategies. This extension requires additional context information for method specifications and requirements, in order to resolve ambiguities that do not occur in single inheritance languages. The combination of healthiness and lazy behavioral subtyping has the advantage that requirements from two independent class hierarchies do not interfere with each other when the hierarchies are combined in a common subclass. This is essential in an incremental proof system.

The inference rules for incremental reasoning presented in this paper are essentially syntax-driven and would form a good basis for integrating behavioral reasoning in a tool supported environment for program development. In such a tool, specifications for method definitions must be manually annotated, whereas method requirements in proof outlines may often be inferred. The integration of lazy behavioral subtyping in the KeY tool [6] is currently being investigated. This integration will allow more elaborate case studies to better evaluate the methodology and practical applicability of the approach.

References

1. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1991.
2. P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'90)*, volume 25(10), pages 161–168. ACM Press, Oct. 1990.
3. K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
4. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 1991.
5. I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.
7. 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*, 7(3):212–232, 2005.
8. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.
9. C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.
10. W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In Necula and Wadler [24], pages 87–99.
11. L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1987.
12. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning for multiple inheritance. Research Report 373, Dept. of Informatics, University of Oslo, Apr. 2008. Available from `http://heim.ifi.uio.no/~creol`.
13. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th Intl. Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer, May 2008.
14. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.

15. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
16. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
17. E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd Intl. Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer, 2005.
18. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
19. S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.
20. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. Rep. 06-20a, Dept. of Comp. Sci., Iowa State University, 2006.
21. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
22. C. Luo and S. Qin. Separation logic for multiple inheritance. *Electronic Notes in Theoretical Computer Science*, 212:27–40, 2008.
23. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2. edition, 1997.
24. G. C. Necula and P. Wadler, editors. *37th Annual Symposium on Principles of Programming Languages (POPL'08)*. ACM, Jan. 2008.
25. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
26. M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In Necula and Wadler [24].
27. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
28. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
29. J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer, July 1996.
30. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.
31. B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.
32. D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA'06)*, pages 345–362. ACM, 2006.