Lazy Behavioral Subtyping[☆]

Johan Dovland[a,*], Einar Broch Johnsen[a], Olaf Owe[a], Martin Steffen[a]

[a]*Department of Informatics, University of Oslo, Norway*

Abstract

Late binding allows flexible code reuse but complicates formal reasoning significantly, as a method call's receiver class is not statically known. This is especially true when programs are incrementally developed by extending class hierarchies. This paper develops a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification is avoided without restricting method overriding to fully behavior-preserving redefinition. The approach ensures that when analyzing the methods of a class, it suffices to consider that class and its superclasses. Thus, the full class hierarchy is not needed, and *incremental* reasoning is supported. We formalize this approach as a calculus which lazily imposes context-dependent subtyping constraints on method definitions. The calculus ensures that all method specifications required by late bound calls remain satisfied when new classes extend a class hierarchy. The calculus does not depend on a specific program logic, but the examples in the paper use a Hoare style proof system. We show soundness of the analysis method. The paper finally demonstrates how lazy behavioral subtyping can be combined with interface specifications to produce an incremental and modular reasoning system for object-oriented class hierarchies.

Key words: behavioral subtyping, object orientation, inheritance, late binding, proof systems, code reuse, method redefinition, incremental reasoning

[☆]This work was done the context of the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (`http://credo.cwi.nl`).

[*]Corresponding author.

Email addresses: `johand@ifi.uio.no` (Johan Dovland), `einarj@ifi.uio.no` (Einar Broch Johnsen), `olaf@ifi.uio.no` (Olaf Owe), `msteffen@ifi.uio.no` (Martin Steffen)

1. Introduction

Late binding of method calls is a central feature in object-oriented languages and contributes to flexible code reuse. A class may extend its superclasses with new methods, possibly overriding the existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the binding of a method call to code cannot be statically determined; i.e., the binding at run-time depends on the actual class of the called object. In addition, object-oriented programs are often designed under an *open world assumption*: Class hierarchies are extended over time as subclasses are gradually developed and added. In general, a class hierarchy may be extended with new subclasses in the future, which will lead to new potential bindings for overridden methods.

To control this flexibility, existing reasoning and verification strategies impose restrictions on inheritance and redefinition. One strategy is to ignore openness and assume a closed world; i.e., the proof rules assume that the complete inheritance tree is available at reasoning time (e.g., [44]). This severely restricts the applicability of the proof strategy; for example, libraries are designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, intended to support incremental development and analysis. If the reasoning relies on the world being closed, extending the class hierarchy requires a costly reverification.

An alternative strategy is to reflect in the verification system that the world is open, but to constrain how methods may be redefined. The general idea is that in order to avoid reverification, any redefinition of a method through overriding must *preserve* certain properties of the method being redefined. An important part of the properties to be preserved is the method's contract; i.e., the pre- and postconditions for its body. The contract can be seen as a description of the promised behavior of all implementations of the method as part of its interface description, the method's *specification*. Best known as *behavioral subtyping* (e.g, [36, 5, 33, 45]), this strategy achieves incremental reasoning by limiting the possibilities for code reuse. Once a specification is given for a method, this specification may not change in later redefinitions. However, behavioral subtyping has been criticized for being overly restrictive and often violated in practice [46]; e.g., it is not respected by the standard Java library definitions.

This paper relaxes the restriction to property preservation which applies in behavioral subtyping, while embracing the open world assumption of incremental program development. The basic idea is as follows: given a method m specified by a precondition p and a postcondition q, there is no need to restrict the behavior of methods overriding m and require that these adhere to that specification. Instead it suffices to preserve the "part" of p and q that is actually *used to verify* the program at the current stage. Specifically, if m is used in the program in the form of a method call $\{r\}\ e.m(\ldots)\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior. It is in fact these weaker conditions that need to be preserved in order to avoid reverification. We call the corresponding analysis strategy *lazy behavioral subtyping*. This strategy may serve as a blueprint for integrating a flexible system for program verification of late bound method calls into environments for object-oriented program development and analysis tools (e.g., [9, 10, 12]).

The paper formalizes the lazy behavioral subtyping analysis strategy using an object-oriented kernel language, based on Featherweight Java [29], and using Hoare style proof outlines. Formalized as a syntax-driven inference system, class analysis is done in the context of a *proof environment* constructed during the analysis. The environment keeps track of the context-dependent requirements on method definitions, derived from late bound calls in the known class hierarchy. The strategy is incremental; for the analysis of a class C, only knowledge of C and its superclasses is needed. We first present a simple form of the calculus, previously published in [21]. In the present paper, the soundness proofs are given for this calculus. Although this system ensures that old proofs are never violated, external calls may result in additional proof obligations in a class which has already been analyzed. As a consequence, it may be necessary to revisit classes at a later stage in the program analysis. To improve this situation, we further extend [21] by considering a refined version of the calculus which introduces behavioral interfaces to encapsulate objects. The requirements of the behavioral interfaces implemented by a class become proof obligations for that class. As a result, the refined calculus is both *incremental* and *modular*: it is no longer necessary to revisit a class due to requirements on calls which occur later during the analysis of unrelated classes. In the refined system, subtyping applies to the inheritance relationship on interfaces, whereas code may be reused more freely in the class hierarchy.

Paper overview. Section 2 introduces the problem of reasoning about late

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

Figure 1: *Syntax for the language OOL*, where C and m are class and method names (of types *Cid* and *Mid*, respectively). Assignable program variables v include fields f and the reserved variable **return** for return values. Expressions e include v, formal parameters x, the reserved variable **this**, and Boolean expressions b.

binding, Section 3 presents the lazy behavioral subtyping approach developed in this paper, and Section 4 formalizes of the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Late Bound Method Calls

2.1. Syntax for an Object-Oriented Kernel Language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard operational semantics (e.g., similar to that of Featherweight Java [29]). The language is named *OOL*, and the syntax is given in Figure 1. We assume a functional language of side-effect free expressions e, including fields f. Vector notation denotes lists; e.g., a list of expressions is written $\overline{e}$. A program P consists of a list $\overline{L}$ of class definitions, followed by a method body. A class extends a superclass, which may be **Object**, with definitions of fields $\overline{f}$, methods $\overline{M}$, and method specifications $\overline{MS}$. For simplicity, we assume that fields have distinct names, that methods with the same name have the same signature (i.e., method overriding is allowed but not overloading), that programs are type-sound so method binding succeeds, and we ignore the types of fields and methods. For classes B and C, $B \leq C$ denotes the reflexive and transitive *subclass relation* derived from class inheritance. If $B \leq C$, we say that B is *below* C and C is *above* B.

A method M takes formal parameters $\overline{x}$ and contains a statement t as its method body where $\overline{x}$ are read-only. The sequential composition of statements t_1 and t_2 is written $t_1; t_2$. The statement $v :=$ **new** $C()$ creates a new object of class C with fields instantiated to default values, and assigns the new reference to v. (In *OOL*, a possible constructor method in the class must be called explicitly.) We distinguish syntactically between *internal calls* and *external calls*. For an internal call $m(\overline{e})$, the method m is executed on self with actual parameters $\overline{e}$. In an external method call $e.m(\overline{e})$, the object e (which may be self) receives a call to the method m with actual parameters $\overline{e}$. The statements $v := m(\overline{e})$ and $v := e.m(\overline{e})$ assign the value of the method activation's **return** variable to v. If m does not return a value, or if the returned value is of no concern, we sometimes use $e.m(\overline{e})$ or $m(\overline{e})$ directly as statements for simplicity. Note that the list $\overline{e}$ of actual parameter values may be empty. There are standard statements for **skip**, conditionals **if** b **then** t **else** t **fi**, and assignments $v := e$. The reserved variable **this** for self reference is read-only. A method specification $m(\overline{x}) : (p, q)$ defines a pre/post specification (p, q) of the method m. For convenience, we let $m(\overline{x}) : (p, q)\{t\}$ abbreviate the combination of the definition $m(\overline{x})\{t\}$ and the specification $m(\overline{x}) : (p, q)$. Specifications of a method m may be given in the class where m is defined or in a subclass. Notice that even if m is not redefined, different subclasses may well have conflicting specifications of m, since internal calls in the body of m may bind differently due to late binding.

2.2. Late Binding

Late binding, or dynamic dispatch, is a central concept of object-orientation and was already present in Simula [15]. A method call is late bound if the method body is selected at run-time, depending on the callee's actual class. Late bound calls are bound to the first implementation found above the actual class. For a method m defined in class C, we say that this implementation is *visible* from a subclass D if a late bound call to m on an instance of D will bind to the implementation in C. Late binding is illustrated in Figure 2: an object of class C_2 executes an inherited method n_1 defined in its superclass C_1 and this method issues a call to a method m defined in both classes. With late binding, the code selected for execution is associated with the first matching m above C_2; i.e., as the calling object is an instance of class C_2, the method m of C_2 is selected and not the one of C_1. If, however, n_1 were executed in an instance of C_1, the late bound invocation of m would be bound to the definition in C_1. Late binding is central for object-oriented

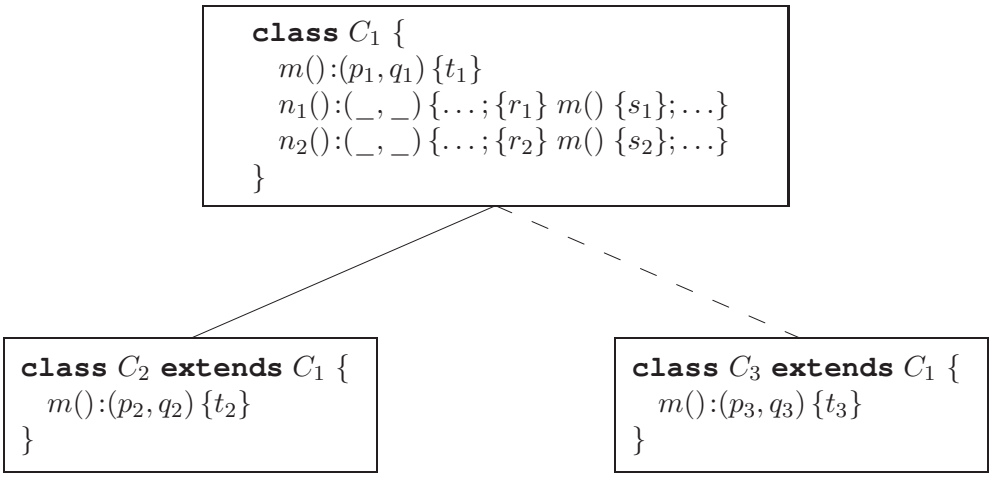

Figure 2: Example of a class hierarchy where the method definitions are decorated with assertions in the style of proof outlines.

programs, and especially underlies many of the well-known object-oriented design patterns [24].

For an internal call to m, made by a method defined in class C, we say that a definition of m in class D is *reachable* if the definition in D is visible from C, or if D is a subclass of C. As m may be overridden by any subclass of C, there may be several reachable definitions for a late bound call statement. For the calls to m in class C_1 in Figure 2, the definitions of m in C_1, C_2, and C_3 are all reachable. At run-time, one of the reachable definitions is selected based on the actual class of the called object. Correspondingly, for an external call $e.m()$ where $e : E$, a definition of m in class D is reachable if the definition is visible from E or if D is a subclass of E.

2.3. Proof Outlines

Apart from the treatment of late bound method calls, our initial reasoning system follows standard proof rules [7, 8] for partial correctness, adapted to the object-oriented setting; in particular, de Boer's technique using sequences in the assertion language addresses the issue of object creation [16]. We present the proof system using Hoare triples $\{p\}\ t\ \{q\}$ [25], where p is the precondition and q is the postcondition to the statement t. The meaning of a triple $\{p\}\ t\ \{q\}$ is standard: if t is executed in a state where p holds and the execution terminates, then q holds after t has terminated. The derivation of

$$\text{(Assign)}\ \{q[e/v]\}\ v := e\ \{q\}$$

$$\text{(New)}\ \{q[\mathbf{new}_C/v]\}\ v := \mathbf{new}\ C()\ \{q\}$$

$$\text{(Skip)}\ \{q\}\ \mathbf{skip}\ \{q\}$$

$$\text{(Seq)}\ \frac{\{p\}\ t_1\ \{r\} \quad \{r\}\ t_2\ \{q\}}{\{p\}\ t_1; t_2\ \{q\}}$$

$$\text{(Cond)}\ \frac{\{p \wedge b\}\ t_1\ \{q\} \qquad \{p \wedge \neg b\}\ t_2\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}\ \{q\}}$$

$$\text{(Adapt)}\ \frac{p \Rightarrow p_1 \quad \{p_1\}\ t\ \{q_1\} \quad q_1 \Rightarrow q}{\{p\}\ t\ \{q\}}$$

$$\text{(Call)}\ \frac{\forall i \in \mathbf{implements}(\mathbf{classOf}(e), m) \cdot \{p_i\}\ body^i_{m(\overline{x})}\ \{q_i\}}{\{\bigwedge_i (p_i[\overline{e}/\overline{x}])\}\ v := e.m(\overline{e})\ \{\bigvee_i (q_i[\overline{e}, v/\overline{x}, \mathbf{return}])\}}$$

Figure 3: *Closed world proof rules.* Let $\mathbf{classOf}(e)$ denote the class of object e and $p[e/v]$ the substitution of all occurrences of v in p by e [25], extended for object creation following [44]. The function $\mathbf{implements}(C, m)$ returns all classes where a call to m from class C may be bound, and $body^i_{m(\overline{x})}$ gives the body of m for class i (assuming that all definitions of m have the same parameter list $\overline{x}$).

triples can be done in any suitable program logic. Let PL be such a program logic and let $\vdash_{PL} \{p\}\ t\ \{q\}$ denote that $\{p\}\ t\ \{q\}$ is derivable in PL. A *proof outline* [41] for a method definition $m(\overline{x})\{t\}$ is an method body decorated with assertions. For the purposes of this paper, we are mainly interested in decorated method calls with pre- and postconditions.

Let the notation $O \vdash_{PL} t : (p, q)$ mean that O is such a proof outline proving that the *specification* (p, q) holds for a body t; i.e., $\vdash_{PL} \{p\}\ O\ \{q\}$ holds when assuming that the pre- and postconditions provided in O for the method calls contained in t are correct. The pre- and postconditions for these method calls are called *requirements*. Thus, for a decorated call $\{r\}\ n()\ \{s\}$ in O, (r, s) is a requirement for n. In order to ensure that this requirement is correct, every reachable definition of n must be analyzed.

2.4. Reasoning about Late Bound Calls in Closed Systems

If the proof system assumes a closed world, all classes must be defined before the analysis can begin, as the requirement to a method call is derived from the specifications of all reachable implementations of that method. To

simplify the presentation in this paper, we omit further details of the assertion language and the proof system (e.g., ignoring the representation of the program semantics — for details see [44]). The corresponding proof system is given in Figure 3; the proof rule (Call) captures late binding under a closed world assumption. The following example illustrates the proof system.

Example 1. Consider the class hierarchy of Figure 2, where the methods are decorated with proof outlines. The specifications of methods n_1 and n_2 play no role in the discussion and are given a wild-card notation $(\_, \_)$. Assume $O_1 \vdash_{PL} t_1 : (p_1, q_1)$, $O_2 \vdash_{PL} t_2 : (p_2, q_2)$, and $O_3 \vdash_{PL} t_3 : (p_3, q_3)$ for the definitions of m in classes C_1, C_2, and C_3, respectively. Consider initially the class hierarchy consisting of C_1 and C_2 and ignore C_3 for the moment. The proof system of Figure 3 gives the Hoare triple $\{p_1 \wedge p_2\}\ m()\ \{q_1 \vee q_2\}$ for each call to m, i.e., for the calls in the bodies of methods n_1 and n_2 in class C_1. In order to apply (Adapt), we get the proof obligations: $r_1 \Rightarrow p_1 \wedge p_2$ and $q_1 \vee q_2 \Rightarrow s_1$ for n_1, and $r_2 \Rightarrow p_1 \wedge p_2$, and $q_1 \vee q_2 \Rightarrow s_2$ for n_2. If the class hierarchy is now *extended* with C_3, the closed world assumption breaks and the methods n_1 and n_2 need to be *reverified*. With the new Hoare triple $\{p_1 \wedge p_2 \wedge p_3\}\ m()\ \{q_1 \vee q_2 \vee q_3\}$ at every call site, the proof obligations given above for applying (Adapt) no longer apply.

3. A Lazy Approach to Incremental Reasoning

This section informally presents the approach of lazy behavioral subtyping. Based on an open world assumption, lazy behavioral subtyping supports incremental reasoning about extensible class hierarchies. The approach is oriented towards reasoning about late bound calls and is well-suited for program development, being less restrictive than behavioral subtyping. A formal presentation of lazy behavioral subtyping is given in Section 4.

To illustrate the approach, first reconsider class C_1 of Example 1. The analysis for n_1 and n_2 requires that $\{r_1\}\ m()\ \{s_1\}$ and $\{r_2\}\ m()\ \{s_2\}$ hold for the internal calls to m in the bodies of n_1 and n_2, respectively. The assertion pairs (r_1, s_1) and (r_2, s_2) may be seen as *requirements* to all reachable definitions of m. Consequently, for m's definition in C_1, both $\{r_1\}\ t_1\ \{s_1\}$ and $\{r_2\}\ t_1\ \{s_2\}$ must hold. Compared to Example 1, the proof obligations for method calls have shifted from the call to the definition site, which allows incremental reasoning. During the verification of a class only the class and its superclasses need to be considered, subclasses are ignored. If we later

analyze subclass C_2 or C_3, the *same requirements* apply to their definition of m. Thus, no reverification of the bodies of n_1 and n_2 is needed when new subclasses are analyzed.

Although C_1 is analyzed independently of C_2 and C_3, its requirements must be considered during the analysis of the subclasses. For this purpose, a *proof environment* is constructed and maintained during the analysis. While analyzing C_1, it is recorded in the proof environment that C_1 requires both (r_1, s_1) and (r_2, s_2) from m. Subclasses are analyzed in the context of this proof environment, and may in turn extend the proof environment with new requirements, tracking the scope of each requirement. For two independent subclasses, the requirements made by one subclass should not affect the other. Hence, the order of subclass analysis does not influence the assertions to be verified in each class. To avoid reverification, the proof environment also tracks the specifications established for each method definition. The analysis of a requirement to a method definition succeeds directly if the requirement follows from the previously established specifications of that method. Otherwise, the requirement may make a new proof outline for the method necessary.

3.1. Assertions and Assertion Entailment

Consider an assertion language with expressions e defined by

$$e ::= \mathbf{this} \mid \mathbf{return} \mid f \mid x \mid z \mid ops(\bar{e})$$

In the assertion language, f is a program field, x a formal parameter, z a logical variable, and *ops* an operation on abstract data types. An *assertion pair* (of type *APair*) is a pair (p, q) of Boolean expressions. Let p' denote an expression p with all occurrences of program variables f substituted by f', avoiding name capture. Since we deal with sets of assertion pairs, the standard adaptation rule of Hoare Logic given in Figure 3 is insufficient. We need an entailment relation which allows us to combine information from several assertion pairs. Consequently, we lift the entailment relation to assertion pairs and to sets of assertion pairs as follows:

Definition 1. (Entailment.) Let (p, q) and (r, s) be assertion pairs and let $\mathcal{U}$ and $\mathcal{V}$ denote the sets $\{(p_i, q_i) \mid 1 \leq i \leq n\}$ and $\{(r_i, s_i) \mid 1 \leq i \leq m\}$. *Entailment* is defined by

1. $(p, q) \rightarrowtriangle (r, s) \triangleq (\forall \overline{z}_1 \;.\; p \Rightarrow q') \Rightarrow (\forall \overline{z}_2 \;.\; r \Rightarrow s')$,
   where $\overline{z}_1$ and $\overline{z}_2$ are the logical variables in (p, q) and (r, s), respectively.

2. $\mathcal{U} \rightarrowtriangle (r, s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall \overline{z}_i \,.\, p_i \Rightarrow q'_i)) \Rightarrow (\forall \overline{z} \,.\, r \Rightarrow s')$.
3. $\mathcal{U} \rightarrowtriangle \mathcal{V} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrowtriangle (r_i, s_i)$.

The relation $\mathcal{U} \rightarrowtriangle (r, s)$ corresponds to classic 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 [7]. Note that when proving entailment, program fields (primed and unprimed) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \rightarrowtriangle \mathcal{V}$.

Example 2. Let x and y be fields, and z_1 and z_2 be logical variables. The assertion pair $(x = y = z_1,\ x = y = z_1 + 1)$ entails $(x = y,\ x = y)$, but it does not entail $(x = z_2,\ x = z_2 + 1)$, since the implication

$$(\forall z_1 \,.\, x = y = z_1 \Rightarrow x' = y' = z_1 + 1) \Rightarrow (\forall z_2 \,.\, x = z_2 \Rightarrow x' = z_2 + 1)$$

does not hold. To see that, consider the program $y := y + 1; x := y$, which satisfies the first assertion pair, but not the second.

Example 3. This example demonstrates entailment for sets of assertion pairs: The two assertion pairs $(x \neq \mathit{null},\ x \neq \mathit{null})$ and $(y = z_1,\ z_1 = \mathit{null} \vee z_1 = y)$ entail $(x \neq \mathit{null} \vee y \neq \mathit{null},\ x \neq \mathit{null} \vee y \neq \mathit{null})$. This kind of reasoning is relevant for reasoning about class invariants without behavioral subtyping: when defining a subclass with a different class invariant than the superclass, the established knowledge of inherited methods may be used to prove the class invariant of the subclass.

3.2. Class Analysis with a Proof Environment

The role of the proof environment during class analyses is now illustrated through a series of examples. The proof environment collects method specifications and requirements in two mappings S and R. Given a class name and a method identifier, these mappings return a set of assertion pairs. The analysis of a class both uses and extends the proof environment.

Propagation of requirements. If the proof outline $O \vdash_{PL} t : (p, q)$ for a method $m(\overline{x})\{t\}$ is derived while analyzing a class C, we extend $S(C, m)$ with (p, q). The requirements on called methods which are encountered during the analysis of O are verified for the known definitions of these methods that are visible from C, and imposed on future subclasses. Thus, for each $\{r\}\ n()\ \{s\}$

in O, the requirement (r, s) must hold for the definition of n that is visible from C. Furthermore, $R(C, n)$ is extended with (r, s) as a restriction on future subclass redefinitions of n.

Example 4. Consider the analysis of class C_1 in Figure 2. The specification (p_1, q_1) is included in $S(C_1, m)$ and the requirements (r_1, s_1) and (r_2, s_2) are included in $R(C_1, m)$. Both requirements must be verified for the definition of m in C_1, since this is the definition of m that is visible from C_1. Consequently, for each (r_i, s_i), $S(C_1, m) \rightarrowtriangle (r_i, s_i)$ must hold, which follows from $(p_1, q_1) \rightarrowtriangle (r_i, s_i)$.

In Example 4, the requirements made by n_1 and n_2 follow from the established specification of m. Generally, the requirements need not follow from the previously shown specifications. In the latter case, it is necessary to provide a new proof outline for the method.

Example 5. If (r_i, s_i) does not follow from (p_1, q_1) in Example 4, a new proof outline $O \vdash_{PL} t_1 : (r_i, s_i)$ must be analyzed similarly to the proof outlines in C_1. The mapping $S(C_1, m)$ is extended by (r_i, s_i), ensuring the desired relation $S(C_1, m) \rightarrowtriangle (r_i, s_i)$.

The analysis strategy must ensure that once a specification (p, q) is included in $S(C, m)$, it will always hold when the definition of method m in C is executed in an instance of any (future) subclass of C, without reverifying m. In particular, when a method n called by m is overridden, the *requirements* made by C must hold for the new definition of n.

Example 6. Consider the class C_2 in Figure 2, which redefines m. By analysis of the proof outline $O_2 \vdash_{PL} t_2 : (p_2, q_2)$, the set $S(C_2, m)$ is extended with (p_2, q_2). In addition, the superclass requirements $R(C_1, m)$ must hold for the new definition of m in order to ensure that the specifications of n_1 and n_2 apply for instances of C_2. Hence, $S(C_2, m) \rightarrowtriangle (r_i, s_i)$ must be ensured for each $(r_i, s_i) \in R(C_1, m)$, similar to $S(C_1, m) \rightarrowtriangle (r_i, s_i)$ in Example 4.

When a method m is (re)defined in a class C, all invocations of m from methods in superclasses will bind to the new definition for instances of C. The new definition must therefore support the requirements from all superclasses.

Let $R{\uparrow}(C, m)$ denote the union of $R(B, m)$ for all $C \leq B$. For each method m defined in C, it is necessary to ensure the following property:

$$S(C, m) \twoheadrightarrow R{\uparrow}(C, m) \tag{1}$$

It follows that m must support the requirements from C itself; i.e., the formula $S(C, m) \twoheadrightarrow R(C, m)$ holds.

Context-dependent properties of inherited methods. Let us now consider methods that are inherited but not redefined. Assume that a method m is inherited from a superclass of a class C. In this case, late bound calls to m from instances of C are bound to the first definition of m above C. However, late bound calls *made by* m are bound *in the context of* C, as C may redefine methods invoked by m. Furthermore, C may impose new requirements on m which were not proved during the analysis of the superclass, resulting in new proof outlines for m. In the analysis of the new proof outlines, we know that late bound calls are bound from C. It would be unsound to extend the specification mapping of the superclass, since the new specifications are only part of the subclass context. Instead, we use $S(C, m)$ and $R(C, m)$ for *local specification and requirement extensions*. These new specifications and requirements only apply in the context of C and not in the context of its superclasses.

Example 7. Let the following class extend the class hierarchy of Figure 2:

```
class C4 extends C1 {
    n():(_, _) {...; {r3} m() {s3}; ...}
}
```

Class C_4 inherits the superclass implementation of m. The analysis of n's proof outline yields $\{r_3\}\ m()\ \{s_3\}$ as requirement, which is included in $R(C_4, m)$ and verified for the inherited implementation of m. The verification succeeds if $S(C_1, m) \twoheadrightarrow (r_3, s_3)$. Otherwise, a new proof outline $O_4 \vdash_{PL} t_1 : (r_3, s_3)$ is analyzed under the assumption that late bound calls are bound in the context of C_4. When analyzed, (r_3, s_3) becomes a specification of m and it is included in $S(C_4, m)$. This mapping acts as a local extension of $S(C_1, m)$ and contains specifications of m that hold in the subclass context.

Assume that a definition of a method m in a class A is visible from C. When analyzing a requirement $\{r\}\ m()\ \{s\}$ in C, we can then rely on $S(A,m)$ and the local extensions of this mapping for all classes between A and C. We assume that programs are type-safe and define a function $S\uparrow$ recursively as follows: $S\uparrow(C,m) \triangleq S(C,m)$ if m is defined in C and $S\uparrow(C,m) \triangleq S(C,m) \cup S\uparrow(B,m)$ otherwise, where B is the immediate superclass of C. We can now revise Equation 1 to account for *inherited methods*:

$$S\uparrow(C,m) \rightarrowtriangle R\uparrow(C,m) \tag{2}$$

Thus, each requirement in $R(B,m)$, for some class B above C, must follow from the established specifications of m in context C. Especially, for each $(p,q) \in R(C,m)$, (p,q) must either follow from the superclass specifications or from the local extension $S(C,m)$. If (p,q) follows from the local extension $S(C,m)$, we are in the case when a new proof outline has been analyzed in the context of C. Note that Equation 2 reduces to Equation 1 if m is defined in C.

Analysis of class hierarchies. A class hierarchy is analyzed in a top-down manner, starting with **`Object`** and an empty proof environment. Classes are analyzed after their respective superclasses, and each class is analyzed without knowledge of its possible subclasses. Methods are specified in terms of assertion pairs (p,q). For each method $m(\overline{x})\{t\}$ defined in a class C, we analyze each (p,q) occurring either as a specification of m, or as an inherited requirement in $R\uparrow(C,m)$. If $S(C,m) \rightarrowtriangle (p,q)$, no further analysis of (p,q) is needed. Otherwise a proof outline O needs to be provided such that $O \vdash_{PL} t : (p,q)$, after which $S(C,m)$ is extended with (p,q). During the analysis of a proof outline, decorated internal calls $\{r\}\ n()\ \{s\}$ yield requirements (r,s) on reachable implementations of n. The $R(C,n)$ mapping is therefore extended with (r,s) to ensure that future redefinitions of n will support the requirement. In addition, (r,s) is analyzed with respect to the implementation of n that is visible from C; i.e., the first implementation of n above C. This verification succeeds immediately if $S\uparrow(C,n) \rightarrowtriangle (r,s)$. Otherwise, a proof outline for n needs to be analyzed in the context of C, which again extends $S(C,n)$ by (r,s). Each call statement in this proof outline is analyzed in the same manner. For *external* calls $\{r\}\ x.m()\ \{s\}$, where x refers to an object of class C', we require that (r,s) follows from the requirements $R\uparrow(C',m)$ of m in C'.

Intuitively, the mapping S reflects the *definition of methods*; each lookup $S(C, m)$ returns a set of specifications for a particular implementation of m. In contrast, the mapping R reflects the *use of methods* and may impose requirements on several implementations.

Lazy behavioral subtyping. Behavioral subtyping in the traditional sense does *not* follow from the analysis method outlined above. Behavioral subtyping enforces the property that whenever a method m is redefined in a class C, its new definition must implement all superclass *specifications* for m; i.e., the method would have to satisfy $S(B, m)$ for all B above C. For example, behavioral subtyping would imply that m in both C_2 and C_3 in Figure 2 must satisfy (p_1, q_1). Instead, the R mapping identifies the requirements imposed by late bound calls. Only these assertion pairs must be supported by overriding methods to ensure that the execution of code from its superclasses does not have unexpected results. Thus, only the behavior assumed by the late bound call statements is ensured at the subclass level. In this way, requirements are *inherited by need*, resulting in a lazy form of behavioral subtyping.

Example 8. Consider a class A defined by

```
class A {
  int n(int y) : (true, return = 5y) {return := 5*y}
  int m(int x) : (x ≥ 0, return ≥ 2x) {return := n(x)}
}
```

By the analysis of method n, the mapping $S(A, n)$ is extended with the specification $(true, return = 5y)$. For the analysis of m, the specification $(x \geq 0, return \geq 2x)$ is included in $S(A, m)$, and let $(y \geq 0, return \geq 2y)$ be the requirement imposed on the internal call to n in the proof outline for m. This requirement is included in the mapping $R(A, n)$. Furthermore, the requirement is verified with regard to the visible definition of n, which succeeds by $S(A, n) \rightarrowtriangle R(A, n)$, i.e.,

$$(true,\ return = 5y) \rightarrowtriangle (y \geq 0,\ return \geq 2y)$$

Next, consider the following extension of A:

```
class B extends A {
  int n(int y) : (true, return = 2y) {return := 2*y}
  int m(int x) : (true, return = 2x)
```

For class B, we include $(true,\ return = 2y)$ in $S(B, n)$, and the inherited requirement $R(A, n)$ is verified with regard to the new implementation of n. This analysis succeeds by $S(B, n) \rightarrowtriangle R(A, n)$, i.e.,

$$(true,\ return = 2y) \rightarrowtriangle (y \geq 0,\ return \geq 2y)$$

Note that behavioral subtyping does not apply to the overriding implementation of n, as the specification $S(A, n)$ cannot be proved for the new implementation. Even though the overriding does not support behavioral subtyping, the verified specification of method m still holds at the subclass level because the requirement imposed by the call to n in the proof outline for m is satisfied by the overriding method.

Class B provides a new specification $(true,\ return = 2x)$ of m, which is included in $S(B, m)$. Analysis of the specification leads to the following requirement: $(true,\ return = 2y) \in R(B, n)$, which follows directly from the specification $S(B, n)$. Note that $(true,\ return = 2x)$ is given as a local extension of the inherited specification of m. Especially, the extension relies on the fact that the internal call to n is bound in the context of class B; the requirement imposed by the call cannot be proven with regard to the implementation of n in A.

4. An Assertion Calculus for Program Analysis

The incremental strategy outlined in Section 3 is now formalized as a calculus $LBS(PL)$ which tracks specifications and requirements for method implementations in an extensible class hierarchy, given a sound program logic PL. Given a program, the calculus builds an environment which reflects the class hierarchy and captures method specifications and requirements. This environment forms the context for the analysis of new classes, possibly inheriting previously analyzed ones. The proof environment is formally defined in Section 4.1, the operations used by $LBS(PL)$ are defined in Section 4.2, and $LBS(PL)$ is given as a set of inference rules in Section 4.3. The soundness of $LBS(PL)$ is established in Section 4.4.

4.1. The Proof Environment of LBS(PL)

A class is represented by a tuple $\langle D, \overline{f}, \overline{M} \rangle$ from which the superclass identifier D, the fields $\overline{f}$, and the methods $\overline{M}$ are accessible by observer functions *inh*, *att*, and *mtds*, respectively. Class names are assumed to be

$$
\begin{array}{ll}
bind_{\mathcal{E}}(nil,\, m) & \triangleq\ nil \\
bind_{\mathcal{E}}(C,\, m) & \triangleq\ \textbf{if}\ m \in P_{\mathcal{E}}(C).mtds\ \textbf{then}\ C\ \textbf{else}\ bind_{\mathcal{E}}(P_{\mathcal{E}}(C).inh,\, m) \\
S{\uparrow}_{\mathcal{E}}(nil, m) & \triangleq\ \emptyset \\
S{\uparrow}_{\mathcal{E}}(C, m) & \triangleq\ \textbf{if}\ m \in P_{\mathcal{E}}(C).mtds\ \textbf{then}\ S_{\mathcal{E}}(C, m) \\
 & \qquad\qquad \textbf{else}\ S_{\mathcal{E}}(C, m) \cup S{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).inh, m) \\
R{\uparrow}_{\mathcal{E}}(nil, m) & \triangleq\ \emptyset \\
R{\uparrow}_{\mathcal{E}}(C, m) & \triangleq\ R_{\mathcal{E}}(C, m) \cup R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).inh, m) \\
body_{\mathcal{E}}(C, m) & \triangleq\ P_{\mathcal{E}}(bind_{\mathcal{E}}(C,\, m)).mtds(m).body \\
C\ \leq_{\mathcal{E}}\ D & \triangleq\ C = D \vee P_{\mathcal{E}}(C).inh\ \leq_{\mathcal{E}}\ D
\end{array}
$$

Figure 4: Auxiliary function definitions

unique, and method names to be unique within a class. The superclass identifier may be *nil*, representing no superclass (for class **`Object`**).

Definition 2. (Proof environments.) A *proof environment* $\mathcal{E}$ of type *Env* is a tuple $\langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}} \rangle$, where $P_{\mathcal{E}} : Cid \rightarrow Class$ is a partial mapping and $S_{\mathcal{E}}, R_{\mathcal{E}} : Cid \times Mid \rightarrow Set[APair]$ are total mappings.

In a proof environment $\mathcal{E}$, the mapping $P_{\mathcal{E}}$ reflects the class hierarchy, $S_{\mathcal{E}}(C, m)$ the set of specifications for m in C, and $R_{\mathcal{E}}(C, m)$ a set of requirements to m from C. For the *empty environment* $\mathcal{E}_{\emptyset}$, $P_{\mathcal{E}_{\emptyset}}(C)$ is undefined and $S_{\mathcal{E}_{\emptyset}}(C, m) = R_{\mathcal{E}_{\emptyset}}(C, m) = \emptyset$ for all $C : Cid$ and $m : Mid$.

Some *auxiliary functions* on proof environments $\mathcal{E}$ are now defined. Let $M.body = t$ for a method definition $M = m(\overline{x})\{t\}$. Denote by $\overline{M}(m)$ the definition of method with name m in $\overline{M}$, by $m \in \overline{M}$ that m is defined in $\overline{M}$, by $t' \in t$ that the statement t' occurs in the statement t, and by $C \in \mathcal{E}$ that $P_{\mathcal{E}}(C)$ is defined. The function $bind_{\mathcal{E}}(C,\, m) : Cid \times Mid \rightarrow Cid$ returns the first class above C in which the method m is defined. Assuming type safety, this function will never return *nil* for well-typed programs. Let the recursively defined functions $S{\uparrow}_{\mathcal{E}}(C, m)$ and $R{\uparrow}_{\mathcal{E}}(C, m) : Cid \times Mid \rightarrow Set[APair]$ return all specifications of m above C and below $bind_{\mathcal{E}}(C,\, m)$, and all requirements to m that are made by all classes above C in the proof environment $\mathcal{E}$, respectively. Finally, $body_{\mathcal{E}}(C, m) : Cid \times Mid \rightarrow Stm$ returns the implementation of m in $bind_{\mathcal{E}}(C,\, m)$. Let $\leq_{\mathcal{E}}$: $Cid \times Cid \rightarrow Bool$ be

the reflexive and transitive subclass relation on $\mathcal{E}$. The definitions of these functions are given in Figure 4.

A *sound environment* reflects that the analyzed classes are correct. If an assertion pair appears in $S_{\mathcal{E}}(C, m)$, there must be a verified proof outline O in PL for the corresponding method body, i.e., $O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q)$. Let n be a method called by m, and let $\overline{x}$ be the formal parameters of n. For all internal calls $\{r'\}\ v := n(\overline{e})\ \{s'\}$ in the proof outline O, (r, s) must be included in $R_{\mathcal{E}}(C, n)$, where $r' = r[\overline{e}/\overline{x}]$, and $s' = s[\overline{e}, v/\overline{x}, \mathbf{return}]$. Thus, all requirements made by the proof outline are in the R mapping. For external calls $\{r'\}\ v := e.n(\overline{e})\ \{s'\}$ in O, where e is of type D, the requirement (r, s) must follow from the requirements of n in the context of D. Note that D may be independent of C; i.e., neither above nor below C. Finally, method specifications must entail the requirements (see Equation 2 of Section 3.2). Sound environments are defined as follows:

Definition 3. (Sound environments.) A *sound environment* $\mathcal{E}$ satisfies the following conditions for all $C : \mathit{Cid}$ and $m : \mathit{Mid}$:

1. $\forall (p, q) \in S_{\mathcal{E}}(C, m)\ .\ \exists O\ .\ O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q)$
   $\wedge\ \forall \{r'\}\ v := n(\overline{e})\ \{s'\} \in O\ .\ R_{\mathcal{E}}(C, n) \twoheadrightarrow (r, s)$
   $\wedge\ \forall \{r'\}\ v := e.n(\overline{e})\ \{s'\} \in O\ .\ e : D \Rightarrow R{\uparrow}_{\mathcal{E}}(D, n) \twoheadrightarrow (r, s)$
2. $S{\uparrow}_{\mathcal{E}}(C, m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(C, m)$

Note that in Condition 1 of Definition 3, the method implementation $\mathit{body}_{\mathcal{E}}(C, m)$ need not be in C itself; the proof outline O may be given for an inherited method definition.

Reverification is avoided by incrementally extending $S_{\mathcal{E}}(C, m)$. If a late bound call requires a verified specification, it is found in $S_{\mathcal{E}}(C, m)$. Thus, the avoidance of reverification can be seen as a dual to the first condition of Definition 3: If $O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q)$ for some proof outline O such that requirements to the method calls in $\mathit{body}_{\mathcal{E}}(C, m)$ follow from the requirement mapping, then the specification (p, q) is added to $S_{\mathcal{E}}(C, m)$.

Let $\models_C \{p\}\ t\ \{q\}$ denote $\models \{p\}\ t\ \{q\}$ under the assumption that internal calls in t are bound in the context of C, and that each external call in t is bound in the context of the actual class of the called object. Let $\models_C m(\overline{x}){:}(p, q)\ \{t\}$ be given by $\models_C \{p\}\ t\ \{q\}$. If there are no method calls in t and $\vdash_{PL} \{p\}\ t\ \{q\}$, then $\models \{p\}\ t\ \{q\}$ follows by the soundness of PL. The following property holds for sound environments:

Lemma 1. *Given a sound environment $\mathcal{E}$ and a sound program logic PL. For all classes $C : Cid$, methods $m : Mid$, and assertion pairs $(p, q) : APair$ such that $C \in \mathcal{E}$ and $(p, q) \in S{\uparrow}_{\mathcal{E}}(C, m)$, we have $\models_D m(\overline{x}){:}(p, q)\,\{body_{\mathcal{E}}(C, m)\}$ for each $D \leq_{\mathcal{E}} C$.*

Proof. By induction on the call structure of m. Since $(p, q) \in S{\uparrow}_{\mathcal{E}}(C, m)$, it follows from the definition of $S{\uparrow}$ in Figure 4 that there exist some class B such that $C \leq_{\mathcal{E}} B$, $bind_{\mathcal{E}}(C, m) = bind_{\mathcal{E}}(B, m)$, and $(p, q) \in S_{\mathcal{E}}(B, m)$. Thus, $body_{\mathcal{E}}(C, m) = body_{\mathcal{E}}(B, m)$. Since $(p, q) \in S_{\mathcal{E}}(B, m)$, there must, by Definition 3, Condition 1, exist some proof outline O such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$.

In this proof outline, each method call is decorated with pre- and post-conditions; i.e., the outline is on the form

$$\{p\}t_0\{r_1\}call_1\{s_1\}t_1\{r_2\}call_2\{s_2\}\ldots\{r_n\}call_n\{s_n\}t_n\{q\}$$

assuming no method calls in the statements $t_0, \ldots, t_n$. For the different t_i, soundness of PL then gives $\models_D \{p\}\ t_0\ \{r_1\}, \models_D \{s_i\}\ t_i\ \{r_{i+1}\}$, and $\models_D \{s_n\}\ t_n\ \{q\}$, for $1 \leq i \leq n-1$. Each call statement is on the form $v := n(\overline{e})$ or $v := e.n(\overline{e})$. For internal calls, we must establish $\models_D \{r_i\}\ v := n(\overline{e})\ \{s_i\}$ as these are bound in context D by lemma assumptions. For external calls, given $e : G$, we must ensure $\models_{G'} \{r_i\}\ v := e.n(\overline{e})\ \{s_i\}$ for any $G' \leq_{\mathcal{E}} G$.

Base case: The execution of $body_{\mathcal{E}}(C, m)$ does not lead to any method calls. Then $\models_D m(\overline{x}){:}(p, q)\,\{body_{\mathcal{E}}(C, m)\}$ follows by the soundness of PL.

Induction step: For each call to some method n in the body of m, bound in context E, assume as induction hypothesis that for all $(g, h) \in S{\uparrow}_{\mathcal{E}}(E, n)$, we have $\models_E n(\overline{y}){:}(g, h)\,\{body_{\mathcal{E}}(E, n)\}$. Internal and external calls are considered separately.

Consider a method call $\{r'\}\ v := n(\overline{e})\ \{s'\}$ in O, and let $r' = r[\overline{e}/\overline{y}]$ and $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$. By the assumptions of the Lemma, the call is bound in the context of class $D \leq_{\mathcal{E}} C$, which means that the induction hypothesis reduces to for all $(g, h) \in S{\uparrow}_{\mathcal{E}}(D, n)$, we have $\models_D n(\overline{y}){:}(g, h)\,\{body_{\mathcal{E}}(D, n)\}$. By Definition 3, Condition 1, we have $R_{\mathcal{E}}(B, n) \twoheadrightarrow (r, s)$. Then $\models_D \{r\}\ n\ \{s\}$ follows since $S{\uparrow}_{\mathcal{E}}(D, n) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(D, n)$ by Definition 3, Condition 2, which especially means $S{\uparrow}_{\mathcal{E}}(D, n) \twoheadrightarrow R_{\mathcal{E}}(B, n)$.

Consider a method call $\{r'\}\ v := e.n(\overline{e})\ \{s'\}$ in O, and let $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $e : E$. From Definition 3, Condition 1, we have $R{\uparrow}_{\mathcal{E}}(E, n) \twoheadrightarrow (r, s)$. The call can be bound in the context of any class E' below E. From Definition 3, Condition 2, we have $S{\uparrow}_{\mathcal{E}}(E', n) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(E', n)$,

which especially means $S{\uparrow_{\mathcal{E}}}(E', n) \twoheadrightarrow (r, s)$. The conclusion $\models_{E'} \{r\}\ n\ \{s\}$ then follows by the induction hypothesis. □

In a *minimal* environment $\mathcal{E}$, the mapping $R_{\mathcal{E}}$ only contains requirements that are caused by some proof outline; i.e., there are no superfluous requirements. Minimal environments are defined as follows:

Definition 4. (Minimal Environments.) A proof environment $\mathcal{E}$ is *minimal* iff

$$\forall (r, s) \in R_{\mathcal{E}}(C, n)\ .\ \exists\, p, q, m, O\ .$$
$$(p, q) \in S_{\mathcal{E}}(C, m) \wedge O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q) \wedge \{r'\}\ v := n(\overline{e})\ \{s'\} \in O.$$

4.2. The Analysis Operations of LBS(PL)

An open program may be extended with new classes, and there may be mutual dependencies between the new classes. For example, a method in a new class C can call a method in another new class D, and a method in D can call a method in C. In such cases, a complete analysis of one class cannot be carried out without consideration of mutually dependent classes. We therefore choose *modules* as the granularity of program analysis, where a module consists of a set of classes. Such a module is *self-contained* with respect to an environment $\mathcal{E}$ if all method calls inside the module can be successfully bound inside that module or to classes represented in $\mathcal{E}$.

In the calculus, judgments have the form $\mathcal{E} \vdash \mathcal{M}$, where $\mathcal{E}$ is the proof environment and $\mathcal{M}$ is a list of *analysis operations* on the class hierarchy. The analysis operations have the following syntax:

$$\begin{array}{lll} \mathcal{O} & ::= \epsilon \mid \mathit{anReq}(\overline{M}) \mid \mathit{anSpec}(\overline{MS}) \mid \mathit{verify}(m, \overline{R}) \mid \mathit{anCalls}(t) \mid \mathcal{O} \cdot \mathcal{O} \\ \mathcal{L} & ::= \emptyset \mid L \mid \mathit{require}(C, m, (p, q)) \mid \mathcal{L} \cup \mathcal{L} \\ \mathcal{M} & ::= \mathit{module}(\overline{L}) \mid [\langle C : \mathcal{O} \rangle\ ;\ \mathcal{L}] \mid [\epsilon\ ;\ \mathcal{L}] \mid \mathcal{M} \cdot \mathit{module}(\overline{L}) \end{array}$$

These analysis operations may be understood as follows. The module operation $\mathit{module}(\overline{L})$ starts the analysis of the classes in the set $\overline{L}$. Classes are assumed to be syntactically well-formed and well-typed. Inside a module, the classes are analyzed in some order, captured by the set $\mathcal{L}$. The operation **`class`** C **`extends`** D $\{\overline{f}\ \overline{M}\ \overline{MS}\}$ initiates the analysis of class C. The operation $[\langle C : \mathcal{O} \rangle\ ;\ \mathcal{L}]$ performs the analysis operations $\mathcal{O}$ in the context of class C *before* operations in $\mathcal{L}$ are considered. Upon completion, the analysis yields a term of the form $[\epsilon\ ;\ \mathcal{L}]$. The analysis of a specific class C may involve the following operations, all inside the context of that class. For each

method defined in C, the operation $anReq(\overline{M})$ initiates the analysis of the requirements imposed by the superclasses of C. The operation $anSpec(\overline{MS})$ analyzes the specifications given by C with regard to the visible implementations of the specified methods. The operation $verify(m, \overline{R})$ verifies the set $\overline{R}$ of assertion pairs with respect to the visible implementation of method m. The operation $anCalls(t)$ analyzes the method calls in the statement t. Since the operation only occurs in the context of a class C, late bound calls are bound in this context.

The operation $require(D, m, (p, q))$ applies to external calls to ensure that m in D satisfies the requirement (p, q). Requirements to external calls are lifted outside the context of the calling class C by this operation, and the verification of requirement (p, q) for m in D is shifted into the set of analysis operations $\mathcal{L}$.

4.3. The Inference Rules of LBS(PL)

Program analysis is initiated by the judgment $\mathcal{E}_\emptyset \vdash module(\overline{L})$, where $\overline{L}$ is a module that is self-contained in the empty environment. Subsequent modules are analyzed in sequential order, such that each module is self-contained with respect to the environment resulting from the analysis of previous modules. The analysis of an individual module is carried out by manipulation of the $module(\overline{L})$ operation according to the inference rules explained below. During the analysis of a module, the proof environment is extended in order to keep track of the currently analyzed class hierarchy and the associated method specifications and requirements. When the analysis of a module is completed, the resulting environment represents a verified class hierarchy. New modules may introduce subclasses of classes which have been analyzed in previous modules. The calculus is based on an open world assumption in the sense that a module is analyzed in the context of previously analyzed modules, but it is independent of subsequent modules.

There are three different *environment updates*; the loading of a new class L into the environment and the extension of the specification and requirement mappings with an assertion (p, q) for a given method m and class C. These are denoted $extP(C, D, \overline{f}, \overline{M}), extS(C, m, (p, q))$ and $extR(C, m, (p, q))$, respectively. Environment updates are represented by the operator $\oplus : Env \times Update \rightarrow Env$, where the first argument is the current proof environment

and the second argument is the environment update, defined as follows:

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

Note that the method specifications $\overline{MS}$ of a given class need not be remembered as a part of the class definition by means of the P mapping, as the class analysis will extend the S mapping with these specifications.

The *inference rules* of the assertion calculus are given in Figure 5. Note that $\mathcal{M}$ represents a list of modules which will be analyzed later, and which may be empty. Rule (NewModule) initiates the analysis of a new module $module(\overline{L})$. The analysis continues by manipulation of the $[\epsilon\,;\,\overline{L}]$ operation that is generated by this rule. For notational convenience, we let $\overline{L}$ denote both a set and list of classes.

Rule (NewClass) selects a new class from the current module, and initiates the analysis of the class in the current proof environment. The premises ensure that a class cannot be introduced twice and that the superclass has *already been analyzed*. The class hierarchy is extended with the new class and the analysis continues by analyzing the specifications of the class by means of the $anSpec(\overline{MS})$ operation, and the requirements imposed on methods defined in the class by means of the $anReq(\overline{M})$ operation. Note that at this point in the analysis, the class has no subclasses in the proof environment. Rule (NewSpec) initiates analysis of a method specification with regard to the visible implementation of the method, and rule (NewMtd) generates a set of requirement assertion pairs for a method implementation. The requirement set is constructed from the superclass requirements to the method.

The rules (ReqDer) and (ReqNotDer) address the verification of a particular requirement with respect to a method implementation. If the requirement follows from the specifications of the method, rule (ReqDer) proceeds with the remaining analysis operations. Otherwise, rule (ReqNotDer) must be applied and a proof of the requirement is needed. In this case a proof outline O for the considered method definition must be provided, such that O establishes the given specification (p, q); i.e., one must prove $O \vdash_{PL} t : (p, q)$ in PL, where t is the method body. The analysis then continues by considering the decorated call statements in O by means of an $anCalls(O)$ operation. Remark that (ReqNotDer) is the only rule which extends the S mapping and which requires a new proof in the program logic PL. The considered requirement leads

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

$$\frac{C \notin \mathcal{E} \qquad D \neq \mathit{nil} \Rightarrow D \in \mathcal{E} \qquad \mathcal{E} \oplus \mathit{extP}(C, D, \overline{f}, \overline{M}) \vdash [\langle C :\ \mathit{anSpec}(\overline{MS}) \cdot \mathit{anReq}(\overline{M})\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\epsilon\,;\, \{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{f}\ \overline{M}\ \overline{MS}\}\} \cup \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewClass})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(m(\overline{x}) : (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewSpec})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, R{\uparrow_{\mathcal{E}}}(P_{\mathcal{E}}(C).\mathit{inh}, m)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(m(\overline{x})\{t\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewMtd})$$

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

$$\frac{O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q) \qquad \mathcal{E} \oplus \mathit{extS}(C, m, (p, q)) \vdash [\langle C :\ \mathit{anCalls}(O) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ReqNotDer})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M} \qquad t \textit{ does not contain call statements}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{Skip})$$

$$\frac{\mathcal{E} \oplus \mathit{extR}(C, m, (p, q)) \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\{p'\}\ v := m(\overline{e})\ \{q'\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{IntCall})$$

$$\frac{e : D \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\,;\, \mathcal{L} \cup \{\mathit{require}(D, m, (p, q))\}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\{p'\}\ v := e.m(\overline{e})\ \{q'\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ExtCall})$$

$$\frac{C \in \mathcal{E} \qquad R{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow (p, q) \qquad \mathcal{E} \vdash [\epsilon\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\epsilon\,;\, \{\mathit{require}(C, m, (p, q))\} \cup \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ExtReq})$$

Figure 5: *The inference system*, where $\mathcal{M}$ is a (possibly empty) list of analysis operations. To simplify the presentation, we let p' and q' denote $p[\overline{e}/\overline{x}]$ and $q[\overline{e}, v/\overline{x}, \texttt{return}]$, respectively.

to a new specification for m with respect to C, and the specification itself is assumed when analyzing the method body. This captures the standard approach to reasoning about recursive procedure calls [26].

Rule (Skip) applies to statements which are irrelevant to the *anCalls* analysis. Rule (IntCall) analyzes the requirement of an internal call occurring in some proof outline. The rule extends the R mapping and generates a *verify* operation which analyzes the requirement with respect to the implementation bound from the current class. The extension of the R mapping ensures that future redefinitions of m must respect the new requirement; i.e., the requirement applies whenever future redefinitions are considered by (NewMtd). Rule (ExtCall) handles external calls on the form $v := e.m(\overline{e})$. The requirement to the external method is removed from the context of the current class and propagated as a *require* operation in the module operations $\mathcal{L}$. The class of the callee is found by type analysis of e, expressed by the premise $e : D$. Rule (ExtReq) can first be applied *after* the analysis of the callee class is completed, and the requirement must then follow from the requirements of this class.

In addition, there are lifting rules concerned with the analysis of set and list structures, and trivial cases. These are given in Figure 6 and explained as follows. Rule (EmpClass) concludes the analysis of a class when all analysis operations have succeeded in the context of the class. The analysis of a module is completed by the rule (EmpModule). Thus, the analysis of a module is completed after the analysis of all the module classes and external requirements made by these classes have succeeded.

The rules (NoReq), (NoMtds), and (NoSpec) apply to the empty requirement set, the empty method list, and the empty specification set, respectively. These rules simply continue the analysis with the remaining analysis operations. Finally, the rules (DecompMtds), (DecompReq), (DecompSpec), and (DecompCalls) flatten (non-empty) methods lists, requirements sets, method specification sets, and statements into separate analysis operations. Note that a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\epsilon\ ;\ \emptyset]$; we call $\mathcal{E}'$ the environment resulting from the analysis of $\mathit{module}(\overline{L})$.

Let $LBS(PL)$ denote the reasoning system for lazy behavioral subtyping based on a (sound) program logic PL, which uses a proof environment $\mathcal{E} : \mathit{Env}$ and the inference rules given in Figure 5 and Figure 6.

$$\frac{\mathcal{E} \vdash [\epsilon\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \epsilon\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{EmpClass}) \qquad \frac{\mathcal{E} \vdash \mathcal{M}}{\mathcal{E} \vdash [\epsilon\ ;\ \emptyset] \cdot \mathcal{M}}\ (\textsc{EmpModule})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, \emptyset) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{NoReq})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\emptyset) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{NoMtds})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\emptyset) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{NoSpec})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, \overline{R_1}) \cdot \mathit{verify}(m, \overline{R_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, \overline{R_1}\ \overline{R_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{DecompReq})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t_1) \cdot \mathit{anCalls}(t_2) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t_1; t_2) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{DecompCalls})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\overline{M_1}) \cdot \mathit{anReq}(\overline{M_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\overline{M_1}\ \overline{M_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{DecompMtds})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\overline{MS_1}) \cdot \mathit{anSpec}(\overline{MS_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\overline{MS_1}\ \overline{MS_2}) \cdot \mathcal{O}\rangle\ ;\ \mathcal{L}] \cdot \mathcal{M}}\ (\textsc{DecompSpec})$$

Figure 6: *The inference system: Lifting rules decomposing list-like structures and handling trivial cases.* Here $\mathcal{M}$ is a (possibly empty) list of analysis operations.

4.4. Properties of LBS(PL)

Although the individual rules of the inference system do not preserve soundness of the proof environment, the soundness of the proof environment is preserved by the successful analysis of a module. This allows us to prove that the proof system is sound for module analysis.

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

Proof. Assume given a sound environment $\mathcal{E}$. We consider each condition in Definition 3 by itself.

Condition 1 of Definition 3 applies to each element (p, q) of $S_{\mathcal{E}}(C, m)$. The proof is by induction over the inference rules, and it suffices to consider rule (ReqNotDer) which is the only rule that extends the S mapping. If (p, q) is included in $S_{\mathcal{E}}(C, m)$, this rule ensures the existence of a proof outline O such that $O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q)$. The analysis then continues with the $\mathit{anCalls}(O)$ operation. For each requirement $\{r'\}\ v := n(\overline{e})\ \{s'\}$ in O, rule (IntCall) extends $R_{\mathcal{E}}(C, n)$ with (r, s), where $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $\overline{y}$ are the formal parameters of n. The relation $R_{\mathcal{E}}(C, n) \twoheadrightarrow (r, s)$ is thereby established, as required by Definition 3. For each requirement $\{r'\}\ v := e.n(\overline{e})\ \{s'\}$ in O where $e : D$, rule (ExtCall) generates an operation $\mathit{require}(D, n, (r, s))$, where $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $\overline{y}$ are the formal parameters of n. Application of rule (ExtReq) thereby establishes $R{\uparrow_{\mathcal{E}}}(D, n) \twoheadrightarrow (r, s)$, as required by Definition 3.

Condition 2 of Definition 3 requires $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(C, m)$ for each m and C. The condition can be proved by induction on the height h of the class hierarchy, starting with classes without superclasses.

Base case: Consider a class $C \in \mathcal{E}$ such that $P_{\mathcal{E}}(C).\mathit{inh} = \mathit{nil}$, i.e., class C is at height $h = 0$. The mapping $R_{\mathcal{E}}(C, m)$ is initially empty so if (p, q) is in $R_{\mathcal{E}}(C, m)$, the rule (IntCall) must have been applied adding the analysis operation $\mathit{verify}(m, (p, q))$ within the context of C. Since this operation succeeds, either (ReqDer) or (ReqNotDer) is applied. The desired relation $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow (p, q)$ must hold if (ReqDer) is applied. If (ReqNotDer) is applied, the mapping $S_{\mathcal{E}}(C, m)$ is extended with (p, q), ensuring $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow (p, q)$.

Induction step: We consider a class $C \in \mathcal{E}$ of height $h{+}1$. As the induction hypothesis, we get $S{\uparrow_{\mathcal{E}}}(C', m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(C', m)$ for all classes $C' \in \mathcal{E}$ of height $\leq h$. Let B be the immediate superclass of C, i.e., $B = P_{\mathcal{E}}(C).\mathit{inh}$. Since B is at height h, we may assume $S{\uparrow_{\mathcal{E}}}(B, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(B, m)$ by the induction hypothesis. There are two cases, depending on whether m is defined in C or not. If $m \notin P_{\mathcal{E}}(C).\mathit{mtds}$ then $S{\uparrow_{\mathcal{E}}}(B, m) \subseteq S{\uparrow_{\mathcal{E}}}(C, m)$, giving $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(B, m)$ by the induction hypothesis. If $m \in P_{\mathcal{E}}(C).\mathit{mtds}$, the method is analyzed with the rule (NewMtd), leading to a *verify* operation on each requirement in $R{\uparrow_{\mathcal{E}}}(B, m)$. The analysis of these *verify* operations ensures $S_{\mathcal{E}}(C, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(B, m)$. Consequently, we have $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}}}(B, m)$ also in this case since $S{\uparrow_{\mathcal{E}}}(C, m) = S_{\mathcal{E}}(C, m)$. In both cases we additionally need to ensure $S{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow R_{\mathcal{E}}(C, m)$. This proof is similar to the base case. □

Theorem 2 (Soundness). *If PL is a sound program logic, then LBS(PL) constitutes a sound proof system.*

Proof. It follows directly from the definition of sound environments that the empty environment is sound. Theorem 1 and Lemma 1 guarantee that the environment remains sound during the analysis of class modules. □

Furthermore, the inference system preserves minimality of proof environments; i.e., only requirements needed by some proof outline are recorded in the $R_{\mathcal{E}}$ mapping.

Lemma 2. *If $\mathcal{E}$ is a minimal environment and $\overline{L}$ is a set of class declarations such that a proof of $\mathcal{E} \vdash module(\overline{L})$ leads to the resulting environment $\mathcal{E}'$, then $\mathcal{E}'$ is also minimal.*

Proof. By induction over the inference rules. For a class C and method m, the rule (IntCall) is the only rule that extends $R_{\mathcal{E}}(C, m)$. In order for the rule to be applied, an operation $anCalls(\{p\}\ v := m(\overline{e})\ \{q\})$ must be analyzed in the context of C for some requirement (p, q) to m. This operation can only have been generated by an application of (ReqNotDer), which guarantees that the requirement is needed by some analyzed proof outline. □

Finally we show that the proof system supports verification reuse in the sense that specifications are remembered.

Lemma 3. *Let $\mathcal{E}$ be an environment and $\overline{L}$ a list of class declarations. Whenever a proof outline O such that $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$ is verified during analysis of some class C in $\overline{L}$, the specification (p, q) is included in $S_{\mathcal{E}}(C, m)$.*

Proof. By induction over the inference rules. The only rule requiring the verification of a proof outline is (ReqNotDer), so it suffices to consider this rule. From the premises of (ReqNotDer) it follows that $S_{\mathcal{E}}(C, m)$ is extended with (p, q) whenever $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$ is verified in PL. □

5. External Specification by Interfaces

In the approach presented so far, each class C provides some specifications of the available methods, inherited or defined, in the form of assertion pairs. These are kept in the S part of the proof environments. Their verification generates R requirements for the late bound internal calls occurring in the

class, which are imposed on subclass redefinitions of the called methods. In a subclass, redefined methods are allowed to violate the S specifications of a superclass, but not the R requirements.

A weakness of $LBS(PL)$ concerns the treatment of external calls (as opposed to internal calls): Reasoning about $e.m(\overline{e})$ where e explicitly refers to a callee of type D, must follow from the R requirements to m that have been established for the class D. Those R requirements, however, are generated from internal calls and may not in general provide suitable external properties. This situation can be improved by letting a class provide R requirements such that reasoning about external calls can be based on those. This means a programmer should be aware of the distinction between S and R requirements, and is able to provide both. Furthermore, mutually dependent classes rely on the *require* operations, which enable delayed reasoning. However, if a *require* operations fails, one must change the provided specifications used and redo the analysis. It is therefore interesting to consider other ways of reasoning about external requirements.

In this section we suggest the use of *behavioral interfaces* as a means to specify and reason about requirements on external method calls. A behavioral interface describes the visible methods of a class and their specifications, 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 runtime 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 an external call $e.m(\overline{e})$ can then be done by relying on the behavioral interface of the object expression e, simplifying the (ExtCall) rule presented above to simply check interface requirements. In this way, *require* operations are no longer needed in the proof system.

In Section 5.1, we define the programming language *IOOL*, which extends

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{implements}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\} \\
& & \mid \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{MS}\}
\end{array}
$$

Figure 7: Syntax for the language *IOOL*, extending the syntactic category L of *OOL* (see Figure 1) with interfaces. The other syntactic categories remain unchanged. Here, I denotes interface names of type *Iid*.

OOL with interfaces. In Section 5.2 we define proof environments of type *IEnv* where interface information is accounted for, and in Section 5.3 we define the calculus $LBSI(PL)$ for reasoning about *IOOL* programs.

5.1. Behavioral Interfaces

We now consider the programming language *IOOL* (see Figure 7), which is given by extending *OOL* with interfaces. A *behavioral interface* consists of a set of method names with signatures and semantic constraints on the use of these methods. In Figure 7, an interface I may extend a list $\overline{I}$ of superinterfaces, and declare a set $\overline{MS}$ of method signatures, where behavioral constraints are given as $(pre, post)$ specifications. An interface may declare signatures of new methods not found in its superinterfaces, and it may declare additional specifications of methods declared in the superinterfaces. The relationship between interfaces is restricted to a form of behavioral subtyping. An interface may extend several interfaces, adding to its superinterfaces new syntactic and semantic constraints. In the sequel, it is assumed that the interface hierarchy conforms to these requirements. The interfaces thus form a type hierarchy: if I' extends I, then I' is a *subtype* of I and I is a *supertype* of I'. Let $\preceq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces under the assumption above. Thus, $I' \preceq I$ if I' equals I or if I' (directly or indirectly) extends I.

An interface I *exports* the methods declared in I or in the superinterfaces of I, with the associated constraints (or requirements) on method use. An object *supports* an interface I if the object provides the methods declared in I and adheres to the specifications imposed by I on these methods. Fields are typed by interfaces; if an object supports I (or a subtype of I) then the object may be referenced by a field typed by I. A class *implements* an interface if its code is such that all instances support the interface. The analysis of the class must ensure that this requirement holds. Objects of different classes may support the same interface, corresponding to different implementations of the interface behavior. Only the methods exported by

$$
\begin{array}{ll}
\mathit{mids}(\emptyset) & \triangleq \emptyset \\
\mathit{mids}(m(\overline{x}) : (p, q)\ \overline{MS}) & \triangleq \{m\} \cup \mathit{mids}(\overline{MS}) \\
\\
\mathit{public}_{\mathcal{E}}(\mathit{nil}) & \triangleq \emptyset \\
\mathit{public}_{\mathcal{E}}(I) & \triangleq \mathit{mids}(K_{\mathcal{E}}(I).\mathit{mtds}) \cup \mathit{public}_{\mathcal{E}}(K_{\mathcal{E}}(I).\mathit{inh}) \\
\mathit{public}_{\mathcal{E}}(I\ \overline{I}) & \triangleq \mathit{public}_{\mathcal{E}}(I) \cup \mathit{public}_{\mathcal{E}}(\overline{I}) \\
\\
\mathit{specs}(\emptyset, m) & \triangleq \emptyset \\
\mathit{specs}(n(\overline{x}) : (p, q)\ \overline{MS}, m) & \triangleq \textbf{if } n = m \textbf{ then } \{(p, q)\} \cup \mathit{specs}(\overline{MS}) \\
 & \qquad\qquad \textbf{else } \mathit{specs}(\overline{MS}) \textbf{ fi} \\
\\
\mathit{spec}_{\mathcal{E}}(\mathit{nil}, m) & \triangleq \emptyset \\
\mathit{spec}_{\mathcal{E}}(I, m) & \triangleq \mathit{specs}(K_{\mathcal{E}}(I).\mathit{mtds}, m) \cup \mathit{spec}_{\mathcal{E}}(K_{\mathcal{E}}(I).\mathit{inh}, m) \\
\mathit{spec}_{\mathcal{E}}(I\ \overline{I}, m) & \triangleq \mathit{spec}_{\mathcal{E}}(I, m) \cup \mathit{spec}_{\mathcal{E}}(\overline{I}, m) \\
\\
I \preceq_{\mathcal{E}} J & \triangleq I = J \vee K_{\mathcal{E}}(I).\mathit{inh} \preceq_{\mathcal{E}} J
\end{array}
$$

Figure 8: Auxiliary function definitions, using space as the list separator.

I are available for external invocations on references typed by I. The class may implement additional *auxiliary* methods for internal use. The interface supported by instances of a class is given by the **`implements`** clause in the class definition (see Figure 7).

5.2. A Proof Environment with Interfaces

As before, classes are analyzed in the context of a proof environment. Let *Interface* denote interface tuples $\langle \overline{I}, \overline{MS} \rangle$, and *IClass* denote class tuples $\langle D, \overline{f}, \overline{M}, \overline{MS} \rangle$. The list of superinterfaces $\overline{I}$ and method specifications $\overline{MS}$ of an interface tuple are accessible by the observer functions *inh* and *mtds*, respectively. The supported interface of a class is accessible by the observer function *impl*. Environments of type *IEnv* are defined as follows.

Definition 5. (Proof environments with interfaces.) A proof environment $\mathcal{E}$ of type *IEnv* is a tuple $\langle P_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}} \rangle$ where $P_{\mathcal{E}} : \mathit{Cid} \to \mathit{IClass}$, $K_{\mathcal{E}} : \mathit{Iid} \to \mathit{Interface}$ are partial mappings and $S_{\mathcal{E}}, R_{\mathcal{E}} : \mathit{Cid} \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$ are total mappings.

For an interface I, let $I \in \mathcal{E}$ denote that $K_{\mathcal{E}}(I)$ is defined, and let $\mathit{public}(I)$ denote the set of method identifiers exported by I; thus, $m \in \mathit{public}(I)$ if m

is declared by I or by a supertype of I. A subtype cannot remove methods declared by a supertype, so $public(I) \subseteq public(I')$ if $I' \preceq I$. If $m \in public(I)$, the function $spec(I, m)$ returns a *set* of type $Set[APair]$ with the behavioral constraints imposed on m by I, as declared in I or in a supertype of I. The function returns a set as a subinterface may provide additional specifications of methods inherited from superinterfaces; if $m \in public(I)$ and $I' \preceq I$, then $spec(I, m) \subseteq spec(I', m)$. These functions are defined in Figure 8.

The definition of sound environments is revised to account for interfaces. In Condition 1, the requirement to an external call must now follow from the interface specification of the called object. Consider a requirement stemming from the analysis of an external call $e.m(\overline{e})$ in some proof outline, where $e : I$. As the interface hides the actual class of the object referenced by e, the call is analyzed based on the interface specification of m. A requirement (r, s) must follow from the specification of m given by type I, expressed by $spec(I, m) \rightarrowtriangle (r, s)$. Furthermore, a third condition is introduced, expressing that a class satisfies the specifications of the implemented interface. If C implements an interface I, the class defines (or inherits) an implementation of each $m \in public(I)$. For each such method, the behavioral specification declared by I must follow from the method specification in the class; i.e., $S{\uparrow}(C, m) \rightarrowtriangle spec(I, m)$.

Definition 6. (Sound environments.) A proof environment $\mathcal{E}$ of type *IEnv* is sound if it satisfies the following conditions for each C : *Cid* and m : *Mid*.

1. $\forall (p, q) \in S_{\mathcal{E}}(C, m)$. $\exists O$. $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$
   $\wedge\ \forall \{r'\}\ v := n(\overline{e})\ \{s'\} \in O$. $R_{\mathcal{E}}(C, n) \rightarrowtriangle (r, s)$
   $\wedge\ \forall \{r'\}\ v := e.n(\overline{e})\ \{s'\} \in O$. $e : I \Rightarrow spec_{\mathcal{E}}(I, n) \rightarrowtriangle (r, s)$
2. $S{\uparrow}_{\mathcal{E}}(C, m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(C, m)$
3. $\forall n \in public_{\mathcal{E}}(I)$. $S{\uparrow}_{\mathcal{E}}(C, n) \rightarrowtriangle spec_{\mathcal{E}}(I, n)$, where $I = P_{\mathcal{E}}(C).impl$

Lemma 1 is adapted to the setting of interfaces as follows:

Lemma 4. *Given a sound environment $\mathcal{E}$: IEnv and a sound program logic PL. For all classes C : Cid, methods m : Mid, and assertion pairs (p, q) : APair such that $C \in \mathcal{E}$ and $(p, q) \in S\uparrow_{\mathcal{E}}(C, m)$, we have $\models_D m(\overline{x}){:}(p, q)\,\{body_{\mathcal{E}}(C, m)\}$ for each $D \leq_{\mathcal{E}} C$.*

Proof. The proof is similar to the proof for Lemma 1, except for the treatment of external calls in the induction step. Let O be a proof outline such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$, where $C \leq_{\mathcal{E}} B$, $bind_{\mathcal{E}}(C, m) = bind_{\mathcal{E}}(B, m)$, and $(p, q) \in S_{\mathcal{E}}(B, m)$. Assume as the induction hypothesis that for any external call to n in O, possibly bound in context E and for all $(g, h) \in S{\uparrow}_{\mathcal{E}}(E, n)$, that $\models_E n(\overline{y}) : (g, h)\, \{body_{\mathcal{E}}(E, n)\}$.

Consider a method call $\{r'\}\ v := e.n(\overline{e})\ \{s'\}$ in O, and let $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $e : I$. From Definition 6, Condition 1, we have $spec_{\mathcal{E}}(I, n) \twoheadrightarrow (r, s)$. Consider some class E where $P_{\mathcal{E}}(E).impl = J$. From Definition 6, Condition 3, we have $S{\uparrow}_{\mathcal{E}}(E, n) \twoheadrightarrow spec_{\mathcal{E}}(J, n)$. If the call to n can bind in context E, then type safety ensures $J \preceq_{\mathcal{E}} I$, giving $spec_{\mathcal{E}}(I, n) \subseteq spec_{\mathcal{E}}(J, n)$. We then have $S{\uparrow}_{\mathcal{E}}(E, n) \twoheadrightarrow spec_{\mathcal{E}}(J, n) \twoheadrightarrow spec_{\mathcal{E}}(I, n) \twoheadrightarrow (r, s)$. By the induction hypothesis, we then arrive at $\models_E \{r\}\ n\ \{s\}$. □

Let $I \in \mathcal{E}$ denote that $K_{\mathcal{E}}(I)$ is defined. We define an operation to update a proof environment with a new interface, and redefine the operation for loading a new class:

$$\begin{array}{rcl} \mathcal{E} \oplus extP(C, D, I, \overline{f}, \overline{M}) & \triangleq & \langle P_{\mathcal{E}}[C \mapsto \langle D, I, \overline{f}, \overline{M}\rangle], K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\ \mathcal{E} \oplus extK(I, \overline{I}, \overline{MS}) & \triangleq & \langle P_{\mathcal{E}}, K_{\mathcal{E}}[I \mapsto \langle \overline{I}, \overline{MS}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \end{array}$$

5.3. The Calculus LBSI(PL) for Lazy Behavioral Subtyping with Interfaces

In the calculus for lazy behavioral subtyping with interfaces, judgments have the form $\mathcal{E} \vdash \mathcal{M}$, where $\mathcal{E}$ is the proof environment and $\mathcal{M}$ is a sequence of interfaces and classes. As before, we assume that superclasses appear before subclasses. This ordering ensures that requirements imposed by superclasses are verified in an incremental manner on subclass overridings. Furthermore, we assume that an interface appears before it is used. More precisely, we assume that whenever a class is analyzed, then the supported interface is already part of the environment, and for each external call statement $v := e.m(\overline{e})$ in the class where $e : I$, the interface I is in the environment. These assumptions ensure that the analysis of a class will not be blocked due to a missing superclass or interface.

As the requirements of external calls are now verified against the interface specifications of the called methods, a complete analysis of a class can be performed without any consideration of other classes. For the revised calculus, it therefore suffices to consider individual classes and interfaces as the granularity of program analysis. The module layer of Section 4 is therefore

$$\frac{I \notin \mathcal{E} \qquad \overline{I} \neq nil \Rightarrow \overline{I} \in \mathcal{E} \qquad \mathcal{E} \oplus extK(I, \overline{I}, \overline{MS}) \vdash \mathcal{P}}{\mathcal{E} \vdash (\textbf{interface}\ I\ \textbf{extends}\ \overline{I}\ \{\overline{MS}\}) \cdot \mathcal{P}}\ (\textsc{NewInt})$$

$$\frac{\begin{array}{c} I \in \mathcal{E} \qquad C \notin \mathcal{E} \qquad D \neq nil \Rightarrow D \in \mathcal{E} \\ \mathcal{E} \oplus extP(C, D, I, \overline{f}, \overline{M}) \vdash \\ \langle C :\ anSpec(\overline{MS}) \cdot anReq(\overline{M}) \cdot intSpec(public_{\mathcal{E}}(I)) \rangle \cdot \mathcal{P} \end{array}}{\mathcal{E} \vdash (\textbf{class}\ C\ \textbf{extends}\ D\ \textbf{implements}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\}) \cdot \mathcal{P}}\ (\textsc{NewClass}')$$

$$\frac{e : I \qquad I \in \mathcal{E} \qquad spec_{\mathcal{E}}(I, m) \twoheadrightarrow (p, q) \qquad \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ anCalls(\{p'\}\ v := e.m(\overline{e})\ \{q'\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{ExtCall}')$$

$$\frac{S\!\uparrow_{\mathcal{E}}(C, m) \twoheadrightarrow spec_{\mathcal{E}}(P_{\mathcal{E}}(C).impl, m) \qquad \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ intSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{IntSpec})$$

$$\frac{\mathcal{E} \vdash \langle C :\ intSpec(\overline{m_1}) \cdot intSpec(\overline{m_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ intSpec(\overline{m_1} \cup \overline{m_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{DecompInt})$$

Figure 9: The extended inference system, where $\mathcal{P}$ is a (possibly empty) sequence of classes and interfaces. Rules (NewClass′) and (ExtCall′) replace (NewClass) and (ExtCall). The other rules of Figure 6 and Figure 5 are unchanged.

omitted. The syntax for analysis operations is given by:

$$\begin{array}{ll} \mathcal{M} ::= \mathcal{P} \mid \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P} & \mathcal{O} ::= \epsilon \mid anReq(\overline{M}) \mid anSpec(\overline{MS}) \mid anCalls(t) \\ \mathcal{P} ::= K \mid L \mid \mathcal{P} \cdot \mathcal{P} & \qquad\quad \mid verify(m, \overline{R}) \mid intSpec(\overline{m}) \mid \mathcal{O} \cdot \mathcal{O} \end{array}$$

The new operation $intSpec(\overline{m})$ is used to analyse the interface specifications of methods $\overline{m}$ with regard to implementations found in the considered class.

For *IOOL*, we define a calculus $LBSI(PL)$, consisting of a (sound) program logic PL, a proof environment $\mathcal{E} : IEnv$, and the inference rules listed in Figure 9. In addition to the rules in Figure 9, $LBSI(PL)$ contains the rules in Figure 5 and Figure 6, except the rules (NewClass), (ExtCall), (NewModule), (EmpModule) and (ExtReq). Rules (NewClass) and (ExtCall) are modified as shown in Figure 9, and rule (ExtReq) is superfluous as the requirements from external calls are analyzed in terms of interface specifications. Rules (NewModule) and (EmpModule) are not needed as modules are removed. For the remaining rules in Figure 5 and Figure 6, we assume that module operations are removed as illustrated by (NewClass′) and (ExtCall′).

Focusing on the changes from Figure 5 and Figure 6, the calculus rules are

outlined in Figure 9. Rule (NewInt) extends the environment with a new interface. No analysis of the interface is needed at this point, the specifications of the interface will later be analyzed with regard to each class that implements the interface. (Recall that interfaces are assumed to appear in the sequence $\mathcal{P}$ before they are used.) The rule (NewClass′) is similar to the rule from $LBS(PL)$, except that an operation *intSpec* is introduced which is used to analyze the specifications of the implemented interface. Rule (ExtCall′) handles the analysis of external calls; here, the requirement of the call is analyzed with regard to the interface specification of the callee. Rule (IntSpec) is used to verify interface specifications, and rule (DecompInt) is used to flatten the argument of *intSpec* operations.

In $LBSI(PL)$, the different method specifications play a more active role when analyzing classes. Method specifications are used to establish interface properties, which again are used during the analysis of external calls. Thus, requirements to external calls are no longer analyzed based on knowledge from the R mapping of the callee. The R mapping is only used during the analysis of internal calls.

Soundness. For soundness of $LBSI(PL)$, Theorem 1 is modified as follows.

Theorem 3. *Let PL be a sound program logic, $\mathcal{E}$:IEnv a sound environment, and L be an interface or a class definition. If a proof of $\mathcal{E} \vdash L$ in $LBSI(PL)$ has $\mathcal{E}'$ as its resulting proof environment, then $\mathcal{E}'$ is also sound.*

Proof. The analysis of a new interface maintains soundness as interfaces are assumed to be loaded in the environment before they are used. For the analysis of a class C, we consider each condition of Definition 6 by itself. The proof of Condition 2 is unchanged from Theorem 1 and it is therefore omitted.

Condition 1 of Definition 6 applies to each element (p, q) of $S_{\mathcal{E}}(C, m)$. The proof is by induction over the inference rules, and it suffices to consider rule (ReqNotDer) which is the only rule that extends the S mapping. If $(p, q) \in S_{\mathcal{E}}(C, m)$, this rule ensures the existence of a proof outline O such that $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$. The analysis then continues with an $anCalls(O)$ operation. For each decorated external call $\{r'\}\ v := n(\overline{e})\ \{s'\}$ in O, rule (IntCall) extends $R_{\mathcal{E}}(C, n)$ with (r, s), where $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $\overline{y}$ are the formal parameters of n. The relation $R_{\mathcal{E}}(C, n) \rightarrowtriangle (r, s)$ is thereby established, as required by Definition 6. For each requirement $\{r'\}\ v :=$

$e.n(\overline{e})$ $\{s'\}$ in O where $e : I$, rule (ExtCall′) ensures $spec_{\mathcal{E}}(I, n) \rightarrowtriangle (r, s)$ as required by Definition 6, where $r' = r[\overline{e}/\overline{y}]$, $s' = s[\overline{e}, v/\overline{y}, \mathbf{return}]$, and $\overline{y}$ are the formal parameters of n.

Condition 3 of Definition 6 concerns the interface I implemented by C, i.e., $P_{\mathcal{E}}(C).impl = I$. For each $m \in public_{\mathcal{E}}(I)$, the relation $S{\uparrow_{\mathcal{E}}}(C, m) \rightarrowtriangle spec_{\mathcal{E}}(I, m)$ must hold. When class C is loaded for analysis by rule (NewClass′), an operation $intSpec(public_{\mathcal{E}}(I))$ is scheduled for analysis in the context of C. For each $m \in public_{\mathcal{E}}(I)$, an $intSpec(m)$ operation is thereby analyzed by rule (IntSpec). Since the analysis succeeds, the desired relation $S{\uparrow_{\mathcal{E}}}(C, m) \rightarrowtriangle spec_{\mathcal{E}}(I, m)$ holds. □

6. Example

In this section we illustrate our approach by a small bank account system implemented by a class *PosAccount* and its subclass *FeeAccount*. The example illustrates how interface encapsulation and the separation of class inheritance and subtyping facilitate code reuse. Class *FeeAccount* reuses the implementation of *PosAccount*, but the type of *PosAccount* is not supported by *FeeAccount*. *Thus, FeeAccount does not represent a behavioral subtype of PosAccount.*

A system of communicating components can be specified in terms of the observable interaction between the different components [11, 27]. In an object-oriented setting with interface encapsulation, the observable interaction of an object may be described by the *communication history*, which is a sequence of invocation and completion messages of the methods declared by the interface (ignoring outgoing calls). At any point in time, the communication history abstractly captures the system state. Previous work [20] illustrates how the observable interaction and the internal implementation of an object can be connected. Expressing pre- and postconditions to methods declared by an interface in terms of the communication history allows abstract specifications of objects supporting the interface. For this purpose, we assume an auxiliary variable h of type $Seq[Msg]$, where Msg ranges over invocation and completion (return) messages to the methods declared by the interface. However, for the example below it suffices to consider only completion messages, so a history h will be constructed as a sequence of completion messages by the empty (ϵ) and right append ($\cdot$) constructor. In [20], the communication messages are sent between two named objects, the caller and the callee. However, for the purposes of this example, it suffices to record only

the completed method identity and its parameters, where **this** is implicitly taken as the callee. Furthermore, the considered specifications are independent of the actual callers. We may therefore represent completion messages by $\langle m(\overline{e}, r)\rangle$, where m is a method name, $\overline{e}$ are the actual parameter values for this method call, and r is the returned value. For reasoning purposes, such a completion message is implicitly appended to the history as a side effect of each method termination. As the history accumulates information about method executions, it allows abstract specification of objects in terms of previously executed method calls.

6.1. Class PosAccount

Let an interface *IPosAccount* support three methods *deposit*, *withdraw*, and *getBalance*. The *deposit* method deposits an amount on the bank account as specified by the parameter value and returns the current balance after the deposit. The *getBalance* method returns the current balance. The *withdraw* method returns *true* if the withdrawal succeeded, and *false* otherwise. A withdrawal succeeds only if it leads to a non-negative balance. The current balance of the account is abstractly captured by the function $\mathit{Val}(h)$ defined by induction over the local communication history as follows:

$$
\begin{array}{ll}
\mathit{Val}(\epsilon) & \triangleq\ 0 \\
\mathit{Val}(h \cdot \langle \mathit{deposit}(x, r)\rangle) & \triangleq\ \mathit{Val}(h) + x \\
\mathit{Val}(h \cdot \langle \mathit{withdraw}(x, r)\rangle) & \triangleq\ \textbf{if } r \textbf{ then } \mathit{Val}(h) - x \textbf{ else } \mathit{Val}(h) \textbf{ fi} \\
\mathit{Val}(h \cdot \textbf{others}) & \triangleq\ \mathit{Val}(h)
\end{array}
$$

In this definition, **others** matches all completion messages that are not captured by any of the above cases. In the interface, the three methods are required to maintain $\mathit{Val}(h) \geq 0$.

```
interface IPosAccount {
  int deposit(nat x) : (Val(h) ≥ 0, return = Val(h) ∧ return ≥ 0)
  bool withdraw(nat x) :
                  (Val(h) ≥ 0 ∧ h = h0, return = Val(h0) ≥ x ∧ Val(h) ≥ 0)
  int getBalance() : (Val(h) ≥ 0, return = Val(h) ∧ return ≥ 0)
}
```

As before, $h_0, b_0, \ldots$ denote logical variables. The interface *IPosAccount* is implemented by a class *PosAccount*, given below. In this class, the balance is maintained by a variable *bal*, and a class invariant expresses that the

balance equals $Val(h)$ and remains non-negative. The class invariant of a class expresses properties in terms of the local fields and communication history. The class invariant must be established by the initial state and be maintained by each public method. Thus, the notation **inv** I below expresses that the assertion pair (I, I) is added to $S(C, m)$ for each public method m in the class. The invariant $bal = Val(h)$ connects the internal state of *PosAccount* objects to their observable behavior, and is needed in order to ensure the postconditions declared in the interface.

```
class PosAccount implements IPosAccount {
  int bal = 0;
  int deposit(nat x) : (true, return = bal) {
    update(x); return := bal
  }
  bool withdraw(nat x) : (bal = b0, return = b0 ≥ x) {
    if (bal >= x) then update(-x); return := true
                  else return := false fi
  }
  int getBalance() : (true, return = bal) {return := bal}
  void update(int v) : (bal = b0 ∧ h = h0, bal = b0 + v ∧ h = h0) {
    bal := bal + v
  }
  inv bal = Val(h) ∧ bal ≥ 0
}
```

Notice that the *update* method is hidden by the interface. This means that the method is not available to objects in the environment, but only for internal calls. The following simple definition of *withdraw* maintains the invariant of the class as it preserves $bal = Val(h)$ and does not modify the balance:

```
bool withdraw(int x) {return := false}
```

However, this implementation does not meet the pre/post specification of *withdraw*, which requires that the method must return *true* if the withdrawal can be performed without resulting in a non-negative balance. Next we consider the verification of the *PosAccount* class.

Pre- and postconditions. The pre- and postconditions given in the *PosAccount* class lead to the inclusion of the following specifications in the S mapping:

$$(\mathit{true},\ \mathbf{return} = \mathit{bal}) \in S(\mathit{PosAccount}, \mathit{deposit}) \quad (3)$$

$$(\mathit{bal} = b_0,\ \mathbf{return} = b_0 \geq x) \in S(\mathit{PosAccount}, \mathit{withdraw}) \quad (4)$$

$$(\mathit{true},\ \mathbf{return} = \mathit{bal}) \in S(\mathit{PosAccount}, \mathit{getBalance}) \quad (5)$$

$$(\mathit{bal} = b_0 \wedge h = h_0,\ \mathit{bal} = b_0 + v \wedge h = h_0) \in S(\mathit{PosAccount}, \mathit{update}) \quad (6)$$

These specifications are easily verified for the method bodies of their respective methods. For *deposit* and *withdraw*, these specifications do not lead to any requirements on *update*.

Invariant analysis. The class invariant is analyzed as a pre/post specification for each public method, i.e., for the methods *deposit*, *withdraw*, and *getBalance*. As a result, the S mapping is extended such that

$$(\mathit{bal} = \mathit{Val}(h) \wedge \mathit{bal} \geq 0,\ \mathit{bal} = \mathit{Val}(h) \wedge \mathit{bal} \geq 0) \in S(\mathit{PosAccount}, m), \quad (7)$$

for $m \in \{\mathit{deposit}, \mathit{withdraw}, \mathit{getBalance}\}$. The methods *deposit* and *withdraw* make internal calls to *update*, which result in the two following requirements:

$$\begin{array}{l} R(\mathit{PosAccount}, \mathit{update}) = \\ \quad \{\ (\mathit{bal} = \mathit{Val}(h) \wedge \mathit{bal} \geq 0 \wedge v \geq 0,\ \mathit{bal} = \mathit{Val}(h) + v \wedge \mathit{bal} \geq 0), \\ \quad\ \ (\mathit{bal} = \mathit{Val}(h) \wedge v \leq 0 \wedge \mathit{bal} + v \geq 0,\ \mathit{bal} = \mathit{Val}(h) + v \wedge \mathit{bal} \geq 0)\ \} \end{array} \quad (8)$$

These requirements follow by entailment from Specification (6).

Interface specifications. The implementation of each method exported through interface *IPosAccount* must satisfy the corresponding interface specification, according to rule (IntSpec). For *getBalance*, it can be proved that the method specification, as given by Specifications (5) and (7), entails the interface specification

$$(\mathit{Val}(h) \geq 0,\ \mathbf{return} = \mathit{Val}(h) \wedge \mathbf{return} \geq 0).$$

The verification of the other two methods follows the same outline, which concludes the verification of class *PosAccount*.

6.2. Class FeeAccount

The interface *IFeeAccount* resembles *IPosAccount*, as the same methods are supported. However, *IFeeAccount* takes an additional *fee* for each successful withdrawal, and the balance is no longer guaranteed to be non-negative. For simplicity we take *fee* as a (read-only) parameter of the interface and of the class (which means that it can be used directly in the definition of *Fval* below). As before, the assertion pairs of the methods are expressed in terms of functions on the local history. Define the allowed overdrafts predicate $AO(h)$ by means of a function $Fval(h)$ over local histories h as follows:

$$
\begin{array}{ll}
AO(h) & \triangleq\ Fval(h) \geq -fee \\
Fval(\epsilon) & \triangleq\ 0 \\
Fval(h \cdot \langle deposit(x,r) \rangle) & \triangleq\ Fval(h) + x \\
Fval(h \cdot \langle withdraw(x,r) \rangle) & \triangleq\ \textbf{if } r \textbf{ then } Fval(h) - x - fee \textbf{ else } Fval(h) \textbf{ fi} \\
Fval(h \cdot \textbf{others}) & \triangleq\ Fval(h)
\end{array}
$$

The interface *IFeeAccount* is declared by

```
interface IFeeAccount(nat fee) {
  int deposit(nat x): (AO(h), return = Fval(h) ∧ AO(h))
  bool withdraw(nat x): (AO(h) ∧ h = h0, return = Fval(h0) ≥ x ∧ AO(h))
  int getBalance(): (AO(h), return = Fval(h) ∧ AO(h)) }
```

Note that *IFeeAccount* is not a behavioral subtype of *IPosAccount*: a class that implements *IFeeAccount* will not implement *IPosAccount*. Informally, this can be seen from the postcondition of *withdraw*. For both interfaces, *withdraw* returns true if the parameter value is less or equal to the current balance, but *IFeeAccount* charges an additional fee in this case.

Given that the implementation provided by the *PosAccount* class is available, it might be desirable to reuse the code from this class when implementing *IFeeAccount*. In fact, only the *withdraw* method needs reimplementation. The class *FeeAccount* below implements *IFeeAccount* and extends the implementation of *PosAccount*.

```
class FeeAccount(int fee)
        extends PosAccount implements IFeeAccount {
  bool withdraw(nat x): (bal = b0, return = b0 ≥ x) {
    if (bal >= x) then update(-(x+fee)); return := true
                  else return := false fi
  }
```

inv $bal = Fval(h) \wedge bal \geq -fee$
}

Remark that the interface supported by the superclass is not supported by the subclass. Typing restrictions prohibit that methods on an instance of *FeeAccount* are called through the superclass interface *IPosAccount*.

Pre- and postconditions. As the methods *deposit* and *getBalance* are inherited without redefinition, the specifications of these methods still hold in the context of the subclass. Especially, Specifications (3), (5), and (6) above remain valid. For *withdraw*, the declared specification can be proved:

$$(bal = b_0,\ \mathbf{return} = b_0 \geq x) \in S(FeeAccount, withdraw) \tag{9}$$

Invariant analysis. The subclass invariant can be proved for the inherited methods *deposit* and *getBalance* as well as for the new definition of the *withdraw* method. From the proof outline for *deposit*, the following requirement on *update* is included in the requirement mapping:

$$(bal = Fval(h) \wedge bal \geq -fee \wedge v \geq 0,\ bal = Fval(h) + v \wedge bal \geq -fee) \in R(FeeAccount, update)$$

This requirement follows from Specification (6) of *update*. The analysis of *withdraw* gives the following requirement on *update*, which also follows from Specification (6):

$$(bal = Fval(h) \wedge bal + v \geq -fee,\ bal = Fval(h) + v \wedge bal \geq -fee) \in R(FeeAccount, update)$$

The invariant analysis leads to the inclusion of the invariant as a pre/post specification in the sets $S(FeeAccount, deposit)$, $S(FeeAccount, withdraw)$, and $S(FeeAccount, getBalance)$, similar to Specification (7).

Interface specification. Now reconsider the method *getBalance*. After analyzing the subclass invariant, the specification set for the method is given as follows:

$$\begin{array}{l} S\!\uparrow\!(FeeAccount, getBalance) = \\ \quad \{(bal = Val(h) \wedge bal \geq 0,\ bal = Val(h) \wedge bal \geq 0), \\ \quad\ (true,\ \mathbf{return} = bal), \\ \quad\ (bal = Fval(h) \wedge bal \geq -fee,\ bal = Fval(h) \wedge bal \geq -fee)\}. \end{array} \tag{10}$$

This specification set that can be assumed to prove the interface specification

$$(\mathit{Fval}(h) \geq -\mathit{fee},\ \mathtt{return} = \mathit{Fval}(h) \wedge \mathit{Fval}(h) \geq -\mathit{fee}) \qquad (11)$$

Specification (11) follows by entailment from Specification (10), using (IntSpec). Note that the superclass invariant is not established by the precondition of Specification (11), which means that the inherited invariant cannot be assumed when establishing the postcondition of Specification (11). The other inherited specification is also needed, expressing that **return** equals *bal*. The verification of the interface specifications for *deposit* and *withdraw* follows the same outline.

7. Related and Future Work

Object-orientation poses several challenges to program logics; e.g., inheritance, late binding, recursive and re-entrant method calls, aliasing, and object creation. In the last years, several programming logics have been proposed, addressing various of these challenges. For example, object creation has been addressed by means of specialized allocation predicates [1] or by encoding heap information into sequences [16]. Numerous proof methods, verification condition generators, and validation environments for object-oriented languages have been developed, including [1, 3, 23, 40, 39, 28, 30, 10]. Java in particular has attracted much interest, with advances being made for different, mostly sequential, aspects and sublanguages of that language. In particular, most such formalizations concentrate on closed systems.

A recent state-of-the-art survey of challenges and results for proof systems and verification in the field of sequential object-oriented programs is given in [32], which also provides further pointers into the literature. In that survey, overloading, dynamic method invocation, and inheritance are mentioned as challenges but are otherwise dealt with rather cursorily; i.e., they are not covered further, although these features are undoubtedly central to today's object-oriented languages. For an overview of verification tools based on the Java modeling language JML, see [12].

Proof systems especially studying late bound methods have been shown to be sound and complete by Pierik and de Boer [44], assuming a closed world. See also [43] for a discussion of (relative) completeness in connection with behavioral subtyping. While proof-theoretically satisfactory, the closed world assumption is unrealistic in practice and necessitates costly reverification when the class hierarchy is extended (as discussed in Section 1).

In order to better support object-oriented design, proof systems should be constructed for incremental (or modular [19]) reasoning. Most prominent in that context are different variations of *behavioral subtyping* [36, 46, 33]. The underlying idea is quite simple: subtyping in general is intended to capture "specialization" and in object-oriented languages, this may be interpreted such that instances of a subclass can be used where instances of a superclass are expected. To generalize this subsumption property from types (such as method signatures) to behavioral properties is the step from standard to behavioral subtyping. The notion of behavioral subtyping dates back to America [4] and Liskov and Wing [35, 36], and is also sometimes referred to as Liskov's substitutability principle. The general idea has been explored from various angles. For instance, behavioral subtyping has been characterized model-theoretically [34, 18] and proof-theoretically [6, 36].

Specification inheritance is used to enforce behavioral subtyping in [19], where subtypes inherit specifications from their supertypes (see also [48] which describes specification inheritance for the language Fresco). Virtual methods [45] similarly allow incremental reasoning by committing to certain abstract properties about a method, which must hold for all its implementations. Although sound, the approach does not generally provide complete program logics, as these abstract properties would, in non-trivial cases, be too weak to obtain completeness without over-restricting method redefinition from the point of view of the programmer. Virtual methods furthermore force the developer to commit to specific abstract specifications of method behavior early in the design process. This seems overly restrictive and lead to less reasoning modularity than the approach as such suggests. In particular, the verification platforms for *Spec*$^\sharp$ [9] and JML [12] rely on versions of behavioral subtyping. Wehrheim [47] investigates behavioral subtyping not in a sequential setting but for active objects. Dynamic binding in a general sense, namely that the code executed is not statically known, does not only arise in object-oriented programs. For instance, Clifton and Leavens [14] use ideas from behavioral subtyping to support modular reasoning in the context of aspect-oriented programs.

The fragile base class problem emerges when seemingly harmless superclass updates lead to unexpected behavior of subclass instances [38]. Many variations of the problem relate to imprecise specifications and assumptions made in super- or subclasses. By making method requirements and assumptions explicit, our calculus can detect many issues related to the fragile base class problem. Subclasses can only rely on requirements made explicit in the

requirement property set of the class. Updates in the superclass must respect these assumptions.

Recently incremental reasoning, both for single and multiple inheritance, has been considered in the context of *separation logic* [13, 42, 37]. These approaches distinguish "static" specifications, given for each method implementation, from "dynamic" specifications used to verify late-bound calls. The dynamic specifications are given at the declaration site, in contrast to our work on lazy behavioral subtyping in which late-bound calls are verified based on call-site requirements. As in lazy behavioral subtyping, the goal is "modularity"; i.e., the goal is to avoid reverification when incrementally developing a program. Complementing the results presented in this paper, we have shown how lazy behavioral subtyping can be used in the setting of multiple inheritance in [22], in which strategies for method binding in multiple inheritance class hierarchies are related to lazy behavioral subtyping.

We currently integrate lazy behavioral subtyping in a program logic for Creol [31, 17], a language for dynamically reprogrammable active objects, developed in the context of the European project Credo. This integration requires a generalization of the analysis to *multiple inheritance* and *concurrent objects*, as well as to Creol's mechanism for *class upgrades*. Creol's type system is purely based on interfaces. Interface types provide a clear distinction between internal and external calls. As shown in this paper, the separation of interface level subtyping from class level inheritance allows class inheritance to exploit code reuse quite freely based on lazy behavioral subtyping, while still supporting incremental reasoning techniques. Classes in Creol may implement several interfaces, slightly extending the approach presented in this paper. It is also possible to let interfaces influence the reasoning for internal calls in a more fine-grained manner, with the aim of obtaining even weaker requirements to redefinitions. We are currently investigating the combination of lazy behavioral subtyping with class upgrades. This combination allows class hierarchies to not only evolve by subclass extensions, but also by restructuring the previously analyzed class hierarchy in ways which control the need for reverification.

8. Conclusion

This paper presents lazy behavioral subtyping, a novel strategy for reasoning about late bound method calls. The strategy is designed to support incremental reasoning and avoid reverification in an open setting, where class

hierarchies can be extended by inheritance. To focus the presentation, we have abstracted from many features of object-oriented languages and presented lazy behavioral subtyping for an object-oriented kernel language based on single inheritance. This reflects the mainstream object-oriented languages today, such as Java and C♯.

Behavioral subtyping has the advantage of providing incremental and modular reasoning for open object-oriented systems, but severely restricts code reuse compared to programming practice. Lazy behavioral subtyping also provides incremental reasoning, but supports significantly more flexible reuse of code. In addition lazy behavioral subtyping provides modularity when combined with interfaces, separating the interface and class hierarchies to provide both subtyping and flexible code reuse. This paper presents both systems with soundness proofs. An extended example of typical code reuse in the banking domain demonstrates how incremental reasoning is achieved by lazy behavioral subtyping in a setting where behavioral subtyping does not apply. Lazy behavioral subtyping appears as a promising framework for controlling a range of desirable changes in the development of object-oriented class hierarchies.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna*, volume 2772 of *Lecture Notes in Computer Science*, pages 11–41. Springer-Verlag, 2003.

[2] ACM. *37th Annual Symposium on Principles of Programming Languages (POPL)*, Jan. 2008.

[3] S. Alagic and S. Kouznetsova. Behavioral compatibility of self-typed theories. In *16th European Conference on Object-Oriented Programming (ECOOP 2002), Malaga, Spain, June 10-14, 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 585–608. Springer-Verlag, 2002.

[4] P. America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.

[5] 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*, pages 60–90. Springer-Verlag, 1991.

[6] 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 (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[7] 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.

[8] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer-Verlag, 1991.

[9] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Intl. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[10] 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-Verlag, 2007.

[11] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer-Verlag, 2001.

[12] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proceedings of FMICS '03*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[13] W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In POPL'08 [2], pages 87–99.

[14] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Mar. 2003. Available as Computer Science Tech. Rep. TR03-01a from ftp://ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf.

[15] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[16] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of Foundations of Software Science and Computation Structure, (FOSSACS'99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[17] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Mar. 2007.

[18] K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. D. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1995.

[19] K. K. Dhara and G. T. Leavens. Forcing behavioural subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996. Also as Iowa State University Tech. Rep. TR-95-20c.

[20] J. Dovland, E. B. Johnsen, and O. Owe. Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects. *Electronic Notes in Theoretical Computer Science*, 203(3):19–34, 2008.

[21] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, May 2008.

[22] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning for multiple inheritance. In M. Leuschel and H. Wehrheim, editors, *Proc. 7th International Conference on Integrated Formal Methods (iFM'09)*, volume 5423 of *Lecture Notes in Computer Science*, pages 215–230. Springer-Verlag, Feb. 2009.

[23] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. ACM, 2001. In *SIGPLAN Notices*.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[25] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[26] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer-Verlag, 1971.

[27] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[28] M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[29] 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.

[30] B. Jacobs and E. Poll. A logic for the Java Modelling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[31] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

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

[33] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.

[34] G. T. Leavens and W. E. Wheil. Reasoning about object-oriented programs that use subtypes. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'90)*, pages 212–223. ACM, 1990. In *SIGPLAN Notices* 25(10).

[35] B. Liskov. Data abstraction & hierarchy. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*. ACM, 1987. In *SIGPLAN Notices* 22(12).

[36] 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.

[37] C. Luo and S. Qin. Separation logic for multiple inheritance. *Electronic Notes in Theoretical Computer Science*, 212:27–40, 2008.

[38] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, 1998.

[39] D. v. Oheimb. *Analysing Java in Isabelle/HOL: Formalization, Type Safety, and Hoare-Logics*. PhD thesis, Technische Universität München, 2001.

[40] D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe (FME 2002)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.

[41] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[42] M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In POPL'08 [2].

[43] C. Pierik and F. S. de Boer. On behavioral subtyping and completeness. In *Proceedings of the ECOOP 2005 workshop on Formal Techniques for Java-like Programs*, 2005.

[44] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.

[45] 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-Verlag, 1999.

[46] 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.

[47] H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 23(2):143–170, 2003.

[48] A. Wills. Specification in Fresco. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.