Reasoning about Inheritance and Unrestricted Reuse in Object-Oriented Concurrent Systems*

Olaf Owe
Department of Informatics, University of Oslo, Norway

March 2016

Abstract

Code reuse is a fundamental aspect of object-oriented programs, and in particular, the mechanisms of inheritance and late binding provide great flexibility in code reuse, without semantical limitations other than type-correctness. However, modular reasoning about late binding and inheritance is challenging, and formal reasoning approaches place semantical restrictions on code reuse in order to preserve properties from superclasses. The overall aim of this paper is to develop a formal framework for modular reasoning about classes and inheritance, supporting unrestricted reuse of code, as well as of specifications. The main contribution is a Hoare-style logic supporting free reuse, worked out for a high-level concurrent object-oriented language. We also show results on verification reuse, based on a combination of Hoare-style logic and static checking. An example illustrates the difference to comparable reasoning formalisms.

1 Introduction

In the setting of object-oriented programs, it is desirable to support modular reasoning, allowing separate reasoning of each class and allowing open programs in the sense that the class hierarchy may be extended downwards. Code reuse is a fundamental property of object-orientation, and flexible reuse implies that a class should not put semantic restrictions on reuse in subclasses. However, modular reasoning with flexible code reuse, late binding, and inheritance is an unsolved challenge. Behavioral subtyping [16] is the most common reasoning approach, restricting subclasses to obey the super-class specifications.

Behavioral subtyping is based on the *substitution principle*, i.e., an object variable declared of class C may at run-time refer to an object of class C or a subclass of C. By exploiting the notion of interfaces, this may be replaced by the *interface substitution principle:* an object variable declared of interface F may at run-time refer to an object supporting F or a subinterface of F. This property can be guaranteed by type checking, but requires that all object variables are declared of an interface, and that interface specifications are respected by a subinterface [20, 13]. Then reasoning about remote calls $o.m(..)$ can be done

*This work was supported by the project *IoTSec* of the Norwegian Research Council. This version of the paper is similar to that of iFM'16, with some corrections and coloring.

using the declared interface of o; however, it does not reduce the restrictions on self calls (or local calls) imposed by behavioral subtyping. Lazy behavioral subtyping [11] relaxes this condition; only behavior that is needed to verify self calls in a superclass must be respected by a subclass redefining the method. This gives added flexibility, allowing a larger class of changes without violating reasoning modularity, but reasoning about free code reuse is still not possible without modifying superclass code or specifications.

Consider two classes A and B such that A is above B (i.e., B inherits A). It may happen that B is not a behavioral subclass of A. Then an object variable x declared of class A may at run-time point to a B object. This would be problematic wrt. reasoning, and unexpected behavior may occur. Such cases are non-trivial to detect [25] – and to solve: If all classes are code-wise as desired, one can either weaken the specification of A or split the class hierarchy, for instance redefining B without inheriting A. In the former case, reasoning made about other classes depending on A must be redone, possibly weakening the specifications of these classes. In the latter case, one is giving up on reuse. Each case has severe draw-backs. In the setting suggested in this paper we use separate hierarchies for reuse and for behavior [3, 7]. Classes A and B must be seen through interfaces. Then the specifications of the classes A and B can be strong (give a strong characterization), and A may implement several interfaces, say I_i, while B may implement some of these. An object variable x of interface I_i can point to a B object if and only if B implements I_i, which is checked by static typing. Thus class reuse is possible even if B does not implement all behavior of A, and the behavior of B can be decided independently of A. Reasoning control is established by verifying each class and its implements clauses.

The contribution of this paper is the development of a reasoning framework allowing reasoning about free code reuse. More specifically, we present a Hoare-style logic for modular reasoning about inheritance, late binding, and free reuse of code and specifications. We build on the general approach of *behavioral interface subtyping* [19]. Each class is only required to satisfy its invariant and interface specifications, as well as any other local specifications given in, or inherited by, the class. This means that a method redefined in a subclass may break the requirements of the superclass, even the minimal requirements imposed in the case of lazy behavioral subtyping; and a subclass need not support the interface(s) of the base class. As opposed to lazy behavioral subtyping, no superclass requirements are imposed on a subclass. The consistency of a class is determined by looking at the class itself, its interfaces, and reused code from superclasses. The main idea of behavioral interface subtyping is that in order to reason about self calls we need to be aware of the class of this object. For each class C we reason about the requirements of that class under the assumption that the class of this is exactly C. Thus if at run-time the class of an executing object is C, we may rely on the reasoning about self calls done in the verification C, and for remote calls we rely on the interface substitution principle. This gives rise to sound reasoning, however, the soundness proof in [19] is presented at an abstract level, without considering a specific calculus presenting the details of self calls and late binding. We present here such a logic. In order to achieve reuse of verification results, we combine the logic and static checking.

We consider the setting of asynchronously communicating concurrent objects. In this setting, verification of systems of concurrent objects can be done compositionally. We build on results for inheritance-fee reasoning [10, 9], avoid-

Pr	::=	$[In^*\ Cl]^+$	program
In	::=	**interface** $F([T\ p]^*)$	
		$[$**extends** $[F(\overline{e})]^+]^?\{S^*\ I\}$	interface declaration
Cl	::=	**class** $C([T\ cp]^*)[$**implements** $[F(\overline{e})]^+]^?$	
		$[$**inherits** $[C(\overline{e})]^+]^?$	
		$\{[T\ w\ [:= e]^?]^*\ [s]^?\ M^*\ S^*\ I\}$	class definition
M	::=	$T\ m([T\ x]^*)\ B\ P$	method definition
S	::=	$T\ m([T\ x]^*)\ P$	method signature
B	::=	$\{[[T\ x\ [:= e]^?]^+;]^?\ [s;]^?$ **return** $e\}$	method body
T	::=	F | Any | Void | Bool | String | Int | Nat | ...	types
v	::=	$x \mid w$	variables (local/field)
e	::=	null | void | this | caller | $v \mid cp \mid f(\overline{e}) \mid (e)$	pure expressions
s	::=	**skip** | $v := e$ | $v :=$ **new** $C(\overline{e})$ | $s; s$	basic statements
		| $v := e.m(\overline{e})$ | $v := C : m(\overline{e})$	remote/static call
		| **await** $v := e.m(\overline{e})$ | **await** e	releasing statements
		| **if** e **then** s $[$**else** $s]^?$ **fi**	if statement
P	::=	$[$ **[** $[A,]^?A$ **]** $]^*$	pre-/postcondition
I	::=	$[$**inv** $A]^?$ $[$**where** $A^+]^?$	invariant specification

Figure 1: Core language syntax. Specification elements are written in blue. F denotes an interface name, C a class name, m a method name, p a formal interface parameter, cp a formal class parameter, w a field, x a method parameter or local variable. We let $[\ ]^*$, $[\ ]^+$ and $[\ ]^?$ denote repeated, repeated at least once, and optional parts, respectively; and $\overline{e}$ is a (possibly empty) expression list. Expressions e are side-effect free. Assertions A are Boolean expressions (possibly quantified) and may refer to the local history **h**. The specification **[** A **]** abbreviates **[** $true, A$ **]** , i.e., *true* is the default precondition.

ing here the complications of futures and recursion. The presented logic is oriented towards automatic verification in the sense that for given class and interface specifications, the generation of verification conditions can be mechanized.

2 A Core Language

For our purposes, we consider a strongly typed, high-level core language inspired by Creol [13]. The syntax is given in Figure 1. Interpreters and compilers exist for version of this language (`http://tools.hats-project.eu/`). A program consists of interfaces and classes. A class C may implement a number of interfaces. Class instances represent concurrent and active objects, while local data structure is defined by data types (syntax not given here). An interface may extend other (super)interfaces, adding declarations of methods, requirements, and invariants. A class may extend a (super)class while adding method definitions-/redefinitions, requirements, and invariants. For simplicity we assume read-only access to method and class parameters, as well as this for referring to the current object, and inside a method, caller for referring to the calling object.

When an interface **extends** another (super)interface, all declarations and

specifications are inherited. When a class **inherits** another (super)class, all code and specifications are inherited unless redefined, i.e., a pre/post pair is inherited unless another is stated, and an invariant is inherited unless another is stated, and a method body is inherited unless the method is redefined. Likewise, the implementation clause of a superclass is inherited unless a new implementation clause is provided. Thus a subclass need not support all the interfaces supported by the superclass, nor respect the superclass invariant. Note that this is different from other specification/reasoning frameworks such as Eiffel, *Spec#*[4], JML [5], and Boogie [15].

The language obeys the interface substitution principle, guaranteed through type checking [20, 13]. Object variables must be typed by interfaces (as opposed to classes). A remote call $v := o.m(\overline{e})$ is type correct if the interface of o supports a method m such that the types of the actual parameters $\overline{e}$ are subtypes of (or equal to) those of the formal parameters of m, and the result type of m is a subtype of v. The self call $v :=$ this$.m(\overline{e})$ is allowed when the class of this supports a method m. For simplicity we assume type correctness, and assume that a class does not offer multiple method declarations with the same name. (Otherwise, we could index the method name by the input and output types.) We assume late binding of methods called by dot-notation, i.e., for an object o of run-time class C the execution of a remote call $o.m(...)$ binds to the definition of m in C, if any, or else that of the closest superclass with a definition of m. Similarly, the self call this$.m(\overline{e})$ binds to the closest superclass B with a definition of m, starting with the the run-time class of this. This definition of m is denoted $B : m$. The notation $B : m$ may also be used in program code, resulting in static binding to a superclass B (or above). We distinguish between *exported methods*, those exported through an interface of the class, and *private methods*, those not exported through any interface of the class. Private methods must be called by the notations this$.m$ (dynamic) or $B : m$ (static).

Each object o has its own virtual processor and executes methods calls with o as callee, and has a process queue with method instantiations caused by incoming calls along with suspended method executions. An await statement puts the current method execution on the object's process queue, allowing an enabled process to continue. A *conditional await statement*, **await** c, is enabled when the condition c is enabled, and an *await call statement*, **await** $v := o.m(\overline{e})$, is enabled when the result of the remote call has arrived. In contrast, the current method must wait while a *blocking call*, $v := o.m(\overline{e})$, is executed, unless $o =$ this, in which case the call is executed as a normal stack-based local call.

Specifications are given by means of invariants, pre/post specifications of methods, and implementation clauses. The class invariant must hold in all states exposed through an interface, i.e., it must hold at suspension points and end of public methods. Methods may be specified by pre/post specifications. This is needed for reasoning about self calls, in particular blocking self calls, when the class invariant may be temporary violated. Multiple pre/post specifications of a method are allowed, specifying complementary properties (see Section 3), and a class may implement multiple interfaces. A class not stating nor inheriting an **implements** clause, implements the empty interface Any, which is the superinterface of all interfaces (and with no requirements).

Inheritance. To describe inheritance more precisely, we look at which items are defined in a class and which items are inherited by a class, and from which superclass. Let B be the direct superclass of C. Inheritance of methods, specifications, and interfaces are explained by the semantic functions

- $\mathit{bind}(C, m) = C$, if C defines a body for m, otherwise $\mathit{bind}(B, m)$.
- $\mathit{spec}(C, m) = C$, if C has a pre/post specification of m, otherwise $\mathit{spec}(B, m)$.
- $\mathit{inv}(C) = C$, if C defines an invariant, otherwise $\mathit{inv}(B)$.
- $\mathit{face}(C) = C$, if C includes an implements clause, otherwise $\mathit{face}(B)$.

These functions are partial, being undefined if no superclass has the required item. Inheritance corresponds to point-wise updates of the semantic functions. For instance, $\mathit{spec}(B, m)$ may be overridden by $\mathit{spec}(C, m)$ even if $\mathit{bind}(B, m)$ is not, and $\mathit{spec}(C, m)$ may even violate $\mathit{spec}(B, m)$. In addition all fields are inherited (if names clash, we use the class name to qualify).

History-Based Specification. The local history $\mathbf{h}$ of a class/interface is the time sequence of communication events observed by this object, including

- method calls made by this object, denoted $\mathsf{this} \rightarrow o.m(\overline{e})$
- method calls received by this object, denoted $o \twoheadrightarrow \mathsf{this}.m(\overline{e})$
- method returns made by this object, denoted $o \leftarrow \mathsf{this}.m(\overline{e}; e)$
- method returns received by this object, denoted $\mathsf{this} \twoheadleftarrow o.m(\overline{e}; e)$, and
- creation events made by this object, denoted $\mathsf{this} \rightarrow o.\,\mathbf{new}\, C(\overline{e})$

where o represents the other part in the communication. Note that these events are not visible to o, when $o \neq \mathsf{this}$. Thus the local histories of different objects are by definition disjoint. In the example of this paper histories will only be concerned about method completions, i.e., $\leftarrow$ and $\twoheadleftarrow$ events.

Sequence notation: A sequence $\mathbf{h}$ is either *empty* or of the form $\mathbf{h}; x$ where x is the last element. The notation $\mathbf{h}/s$ denotes the projection of $\mathbf{h}$ restricted to elements in the set s, $\#$ denotes sequence length, and x **before** x' **in** $\mathbf{h}$ denotes that x appears before x' in $\mathbf{h}$, i.e., $\#(\mathbf{h}'/\{x\}) \leq \#(\mathbf{h}'/\{x'\})$ for any sequence prefix $\mathbf{h}'$ of $\mathbf{h}$. For a local history $\mathbf{h}$ we let $\mathbf{h}/F$ denote the projection to the *alphabet* of F, given by events of the form $\mathsf{this} \rightarrow o.\,\mathbf{new}\, C(\overline{e})$, $\mathsf{this} \rightarrow o.m(\overline{e})$, and $\mathsf{this} \twoheadleftarrow o.m(\overline{e}; e)$, as well as events of the form $o \twoheadrightarrow \mathsf{this}.m(\overline{e})$ and $o \leftarrow \mathsf{this}.m(\overline{e}; e)$ for m offered by F. Similar notation applies to classes C, thus $\twoheadrightarrow \mathsf{this}.m$ and $\leftarrow \mathsf{this}.m$ events are restricted to methods defined or inherited in the class.

For a global history H we have that

$$\begin{array}{llll} (o \rightarrow o'.m(\overline{e})) & \mathbf{before} & (o \twoheadrightarrow o'.m(\overline{e})) & \mathbf{in}\, H \\ (o \twoheadrightarrow o'.m(\overline{e})) & \mathbf{before} & (o \leftarrow o'.m(\overline{e}; e)) & \mathbf{in}\, H \\ (o \leftarrow o'.m(\overline{e}; e)) & \mathbf{before} & (o \twoheadleftarrow o'.m(\overline{e}; e)) & \mathbf{in}\, H \end{array}$$

which is formalized by the wellformedness predicate used in the compositional rule for global reasoning [27], which expresses that the global invariant is the conjunction of the wellformedness predicate and all object interface invariants.

Since the alphabets of the objects are by definition disjoint, the wellformedness predicate is needed to connect the different object invariants.

The invariant of a class C may refer to fields, $\mathbf{h}$, and constants, including this. The invariant must be maintained by each non-private method of the class (possibly inherited), and a class must satisfy each implemented interface. A method specification may in addition refer to the formal parameters (including the caller) and the result (return). When seen from another class with a larger alphabet, a C invariant must hold on the original alphabet of C.

The invariant of an interface F may refer to the local history $\mathbf{h}$ (and the constant this) but not fields since these are not visible at the interface level. An invariant $I(\mathbf{h})$ of an interface F is understood as $I(\mathbf{h}/F)$ in a subinterface or class. Thus we define $I_F(\mathbf{h})$ as $I(\mathbf{h}/F)$, and similarly for classes. Abstract variables can be expressed by abstraction functions (say F) over the history, typically by inductive definitions with left hand sides of the the form $F(empty)$ and $F(\mathbf{h}; e)$ for each kind of event e, as demonstrated in the example below.

A Small Example

Figure 2 defines a class BANK, a subclass BANKPLUS, and related interfaces, illustrating typical code reuse, adding complexity to a simple base class. The purpose of the (somewhat contrived) private method *upd* is to demonstrate the difference between non-lazy and lazy behavioral subtyping. The subclass does not respect the base class specification. Similar complications arise when adding transaction fees or interest calculations, while other extensions, such as adding a transaction history, would respect the base class specification. Code reuse is clearly useful both when base class specifications are respected and not.

Interface *Bank* states that the balance (as returned by *bal*) is the sum of amounts deposited (by *add*) or withdrawn (by *sub*) from the bank account, ignoring unsuccessful *add* and *sub* calls, and that *add* calls always succeed. Interface *PerfectBank* extends *Bank* by stating that all *sub* calls succeed, while interface *BankPlus* extends *Bank* by stating that balance is always non-negative.

Interface and type names are capitalized while class names are in upper case letters. The specification of interface Bank illustrates history-based specification. The abstraction function *sum* calculates the balance from the local history. Note that only method return events are used in the specification. In the inductive definition of *sum*, **others** is used to match other cases, and underscore (_) is used to match any expression. The keyword **inv** identifies invariants and **where** identifies auxiliary function definitions. In assertions, **inv** refers to the current invariant, while C: **inv** refers to the invariant of class C.

The class BANK uses a private method *upd* called by both *add* and *sub*. The *upd* method is specified by two complementary pre/post pairs, each specifying a property of the method. The invariant says that the value of the field *bal* is the *sum* calculated over the local history. The subclass BANKPLUS inherits the pre/post specifications of *bal* and *add* from BANK, but not the ones for *upd* and *sub*, which are redefined and therefore not inherited. In fact the subclass violates the pre/post specifications for *upd* and *sub* in BANK. Likewise the **implements** clause is redefined and therefore not inherited. In this example, the subclass does not obey the requirements imposed by behavioral subtyping, since BANKPLUS violates the BANK interface *PerfectBank*, nor by lazy behavioral subtyping since

```
interface Bank { Bool sub(Nat x)
 Bool add(Nat x) [return = true]
 Int bal() [return = sum(h)]
 where sum(empty) = 0, -- sum calculates the balance
  sum(h; (_←this.add(x;true))) = sum(h)+x,
  sum(h; (_←this.sub(x;true))) = sum(h)−x,
  sum(h;others) = sum(h) }

interface PerfectBank extends Bank { Bool sub(Nat x) [return = true]}

interface BankPlus extends Bank { inv sum(h)>=0 }

class BANK implements PerfectBank { Int bal:=0;
 Bool upd(Int x){bal:=bal+x; return true} [return = true]
                    [inv, bal=sum(h)+x and return = true]
 Bool add(Nat x){Bool ok; ok:=this.upd(x); return ok} [return = true]
 Bool sub(Nat x){Bool ok; ok:=this.upd(−x); return ok} [return = true]
 Int bal(){return bal} [return = bal]
 inv bal = sum(h) }

class BANKPLUS implements BankPlus inherits BANK{
 Bool upd(Int x){Bool ok:=(bal+x>=0);
   if ok then ok:=BANK:upd(x)fi; return ok}
            [inv, bal>=0 and bal = sum(h)+if return then x else 0]
            [bal' = bal, return = (bal'+x>= 0)]
 Bool sub(Nat x) [bal' = bal, return = (bal'>= x)] -- renewed specification
 inv BANK:inv and bal >=0 }

class CLIENT { Seq[String] paid; Bank acc; acc:= new BANK;
 Bool salary(Nat x){Bool ok; ok:=acc.add(x); return ok}
 Bool bill(String kid, Nat x, Bank y){ Bool ok:=false;
  if not kid in paid then await ok:=acc.sub(x);
   if ok then y.add(x); paid:=(paid;kid) fi fi; return ok }
 inv paid = p(h) -- p gives the sequence of successful bill payments
 where p(empty) = empty,
  p(h; (_←this.bill(k,x,y;true))) = (p(h); k),
  p(h; others) = p(h) }
```

Figure 2: A bank example, violating behavioral and lazy behavioral subtyping.

BANKPLUS violates the BANK postcondition of *upd*, which is needed for the local *upd* calls in the verification of BANK.

For the sake of completeness a client class is included, showing also blocking and non-blocking calls. The CLIENT invariant expresses that paid corresponds to successful bill payments, calculated over the history by $p(\mathbf{h})$, defined inductively.

3 Hoare-Style Reasoning

The considered core language is chosen with respect to simplicity of semantics, avoiding the complexity of shared variables and low-level synchronization primitives. We consider partial correctness, expressed by Hoare triples of the form

$[P]\ S\ [Q]$, meaning that the condition Q holds in any post-state of the statement s provided the condition P holds in the pre-state [12]. The language satisfies the classical Hoare axiom for assignment

$$\vdash [Q_e^x]\ x := e\ [Q]$$

since there are no side-effects of expressions, remote access to fields, nor shared variables (even though object variables give rise to aliasing). Here Q_e^x denotes Q with all (free) occurrences of x replaced by e. Rules for skip and if-statements are standard, and so is the rule for sequential composition (see Figure 3), because there is no interference between objects since their local conditions are on disjoint variables. In particular the histories of two objects do not share events.

Late binding implies that a method call may behave differently depending on the class of the executing object. Also calls binding to the same body may behave differently since self calls in the body may depend on the class of the executing object. For instance in the *Bank* example a call to *sub* binds to BANK:*sub* (regardless of the class of the executing object), but the *upd* call in the body of BANK:*sub* binds to either BANK:*upd* or BANKPLUS:*upd* depending on the class of the executing object, BANK or BANKPLUS, respectively. For B above C, we use the notation $body_{C::B:m}$ to refer to the execution of the body of m in class B (or above) when this object is of class C. A late bound self call $\mathsf{this}.m(..)$ binds to $body_{C::C:m}$, and the static call $B : m(..)$ binds to $body_{C::B:m}$, and both are executed as a stack-based local call. Given that class $bind(B, m)$ contains a method definition $m(\overline{x})\{s;\ \mathbf{return}\ e\}$, we define $body_{C::B:m}$ by

$$\mathbf{h} := (\mathbf{h}; \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x})); s; \mathsf{return} := e; \mathbf{h} := (\mathbf{h}; \mathsf{caller} \twoheadleftarrow \mathsf{this}.m(\overline{x}; \mathsf{return})) \quad (1)$$

which incorporates the appropriate effects on the local history. It follows that

$$body_{C::B:m} = body_{C::bind(B,m):m}$$

And we have $body_{C::B:m} = body_{B::B:m}$ if the execution of the former body does not lead to suspension nor self calls below B. Such equivalences can be detected (underestimated) by static checking following each execution path of the body of m, following static calls and remote calls where the callee might be this. Such equivalences can be exploited for verification reuse, as shown in the example.

Type checking ensures that binding succeeds, i.e., for all method calls in a type correct program each call binds to a body, apart from remote calls on object variables that are null. Note that calls to null may appear in the history, allowing specifications about the absence of such calls. We let $body_{C::C:init}$ denote the initialization code of C.

Verification of a Class. According to the idea of behavioral interface subtyping, each class C is verified separately under the assumption that the class of this object is exactly C. A major complication is that reasoning about reused code from a superclass is in general different than the reasoning made in the superclass. Hoare-style reasoning must be done relative to the class C of this object. We use the notation $\vdash_C [P]\ s\ [Q]$, where C represent the class of this object. And $\vdash_C Q$ means that the assertion Q can be proved in the context

assign $$\vdash [Q^x_e]\ x := e\ [Q]$$

await guard $$\vdash_C [I_C \wedge L \wedge h' = \mathbf{h}]\ \texttt{await}\ b\ [b \wedge I_C \wedge L \wedge h' \leq \mathbf{h}]$$

new $$\vdash [\forall v' \,.\, \mathit{fresh}(v', \mathbf{h}) \Rightarrow Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this}\to v'.\,\mathbf{new}\,C(\overline{e}))}]\ v := \texttt{new}\ C(\overline{e})\ [Q]$$

blocking call $$\vdash [\forall v' \,.\, o \neq \mathsf{this} \wedge Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this}\to o.m(\overline{e}));(\mathsf{this}\leftarrow o.m(\overline{e});v'))}]\ v := o.m(\overline{e})\ [Q]$$

call on this $$\frac{\vdash_C [P]\ v := C : m(\overline{e})\ [Q]}{\vdash_C [P]\ v := \mathsf{this}.m(\overline{e})\ [Q]}$$

self call $$\frac{\vdash_C [P]\ v := \mathsf{this}.m(\overline{e})\ [Q]}{\vdash_C [o = \mathsf{this} \wedge P]\ v := o.m(\overline{e})\ [Q]}$$

static call $$\frac{\vdash_C [P]\ \mathit{body}_{C::B:m}\ [Q^{\mathbf{h}}_{\mathbf{h};(\mathsf{this}\leftarrow \mathsf{this}.m(\overline{x};\mathsf{return}))}]}{\vdash_C [P^{\overline{x},\mathsf{caller},\mathbf{h}}_{\overline{e},\mathsf{this},\mathbf{h};(\mathsf{this}\to \mathsf{this}.m(\overline{e}))} \wedge L]\ v := B : m(\overline{e})\ [Q^{\overline{x},\mathsf{caller},\mathsf{return}}_{\overline{e},\mathsf{this},v} \wedge L]}$$

entailment $$\frac{\vdash_C [P_j]\ s\ [Q_j], \mathrm{all}\, j \in J \qquad \vdash_C (\wedge_{j\in J}[[P_j, Q_j]]) \Rightarrow\ [[P, Q]]}{\vdash_C [P]\ s\ [Q]}$$

if $$\frac{\vdash_C [P \wedge b]\ s\ [Q] \qquad \vdash_C [P \wedge \neg b]\ s'\ [Q]}{\vdash_C [P]\ \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{fi}\ [Q]}$$

Figure 3: Hoare style rules and axioms. Primed variables represent fresh logical variables, $\mathit{fresh}(v', \mathbf{h})$ expresses that v' does not occur in $\mathbf{h}$, L denotes a local assertion, i.e., without occurrences of fields (statically checked). In rule static call we assume for simplicity that neither v nor fields occur in e (otherwise additional logical variables could be used to freeze the prestate values).

of the specification functions available in C. We write $\vdash [P]\ s\ [Q]$ rather than $\vdash_C [P]\ s\ [Q]$ when the class context of s is irrelevant for the reasoning.

The notation $\vdash_C\ B\ :\ m(\overline{x})[P, Q]$ abbreviates $\vdash_C\ [P]\ \mathit{body}_{C::B:m}\ [Q]$, and $\vdash_C m(\overline{x})[P, Q]$ abbreviates $\vdash_C C : m(\overline{x})[P, Q]$. This notation is convenient when considering class specifications given by pre/postconditions. For a condition Q we let the notation Q/F denote $Q^{\mathbf{h}}_{\mathbf{h}/F}$ where F is a class or interface.

Let I_C denote the given invariant of class C. In order to *verify a class* C the following verification conditions must be proved:

1. $\vdash_C I_C \Rightarrow (I_F/F)$, for each invariant I_F of an interface F of class $\mathit{face}(C)$
2. $\vdash_C \mathit{init}()[\mathit{true}, I_C]$ (i.e., the class initialization establishes I_C)
3. $\vdash_C m(\overline{x})[I_C, I_C]$, for each public method m of C (i.e., maintenance of I_C)
4. $\vdash_C m(\overline{x})[I_C \wedge (P/F), Q/F]$, if an interface F of $\mathit{face}(C)$ contains $m(\overline{x})[P, Q]$
5. $\vdash_C m(\overline{x})[P, Q]$, if C contains or inherits $m(\overline{x})[P, Q]$ ($\mathit{bind}(C, m)$ is defined).

Here 1 and 4 ensure that each interface of C is satisfied, 2 and 3 that the class invariant is satisfied, and 5 ensures any additional pre/post specifications of C, including inherited ones. Note that in 4 we assume the class invariant in the precondition of a public method, since calls from other objects are started in an invariant state. Only blocking self calls may start in non-invariant states.

Each class is verified separately in this way (considering inherited superclass code). Together with correct typing of object variables, this ensures that each object variable will satisfy its declared interfaces, and each object of (run-time) class C will satisfy the interfaces of C. This implies that the compositional rule for reasoning about concurrent object systems is sound, see [19]. Furthermore, the reasoning about inherited code ensures that each late bound self call made at run-time will satisfy the pre/post specifications given in C.

Reasoning Rules

Figure 3 presents all rules related to self calls and histories. For a class C we use $\vdash_C$ to express provability in the context of C as explained. For code in class C this corresponds to normal class-based reasoning. For code inherited by C, reasoning about suspension and self calls depends on C, as reflected in Rule static call, keeping the C context when moving to a superclass B. The importance of the context C is evident in the rule for **`await`** where it is essential that we use I_C, and in the call rules, where both C and B are used to get the relevant pre/postconditions. In general the pre/postconditions of a method m vary both with respect to the enclosing class B and the context class C.

Since we allow multiple pre/postconditions of a given method, we need the entailment rule in order to derive implications of multiple pre/postconditions, using the *relational meaning* of a pre/postcondition $[P, Q]$ given by

$$[[P, Q]] \triangleq \forall \overline{z} \,.\, P^{\overline{w},\mathbf{h}}_{\overline{w}_{in},\mathbf{h}_{in}} \Rightarrow Q^{\overline{w},\mathbf{h}}_{\overline{w}_{out},\mathbf{h}_{out}}$$

where $\overline{z}$ is the list of logical variables in $[P, Q]$, x_{in} denotes the pre-state ("*in*") value of a variable x, and x_{out} denotes the post-state ("*out*") value of x. (Constants including parameters, this, and return are not quantified nor substituted.) For instance, from the two pre/postconditions $[bal \geq x, \mathsf{return} = true]$ and $[bal < x, \mathsf{return} = false]$ we may derive $[bal' = bal, \mathsf{return} = bal' \geq x]$. And the standard consequence rule can be derived from the entailment rule.

The effects on $\mathbf{h}$ from the side of a caller are reflected in the call rules, whereas the effects on $\mathbf{h}$ from the side of the callee are reflected in the definition of *body*, see (1). Self calls give rise to both effects. Rule new is similar, with the additional requirement that the generated object is locally fresh. (Global uniqueness follows by including the parent object in the identity.) According to Rule call on this, the call $v := \mathsf{this}.m(\overline{x})$ is equivalent to the static call $v := C : m(\overline{x})$ where C is the class of this object. The rules self call and blocking call treat blocking calls according to whether the callee equals this. For a call such that the premise of rule self call would not be type correct, we may conclude that $o \neq \mathsf{this}$. This can be formalized by letting the type analysis rewrite a call $v := o.m(\overline{e})$ to $v := o..m(\overline{e})$ whenever the call $v := \mathsf{this}.m(\overline{e})$ is not type correct, and adding the Hoare axiom

$$\vdash [\forall v' \,.\, Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this}\rightarrow o.m(\overline{e}));(\mathsf{this}\leftarrow o.m(\overline{e};v'))}]\; v := o\,..\,m(\overline{e})\; [Q]$$

for such "external calls" (syntactically indicated by ".."), to improve reasoning. This static analysis of object disjointness may be strengthened, for instance by considering static parent-child connections or ownership.

Reasoning about a suspending call **await** $v := o.m(\overline{e})$ is equivalent to reasoning about the pseudo-code

$$\mathbf{h} := (\mathbf{h}; \mathsf{this} \rightarrow o.m(\overline{e})); \ \mathbf{await}\ \mathit{true}; v' := \mathrm{some}; \mathbf{h} := (\mathbf{h}; \mathsf{this} \twoheadleftarrow o.m(\overline{e}; v')); v := v'$$

where "some" represents a non-deterministic value, i.e., $[\forall x \,.\, Q]\ x := \mathrm{some}\ [Q]$. When o is this and m is non-public, one must add the premise $\vdash_C m(\overline{x})[I_C, I_C]$ in order to ensure that the self call preserves the invariant of C.

New properties of a method $B\!:\!m$ can be derived from old properties using the entailment rule or by analysis of $\mathit{body}_{C::B:m}$. Entailment is useful at the level of method specifications since it is natural to keep track of verified properties at this level. The proof of a pre/post specification of m in C will be based on the invariant of C, which may differ from that of B. Thus a pre/post specification of m in B cannot in general be guaranteed in a subclass C. The static call rule reflects this by referring to both B and C.

Since each class is analyzed separately, typically in the order defined, we obtain an open world and modular verification system. In the analysis of a class C we may need to consider superclasses of C, but not subclasses. We may reuse superclass verification results as follows: For code inherited from a superclass B, we may derive $\vdash_C B : m(\overline{x})[P, Q]$ from $\vdash_B B : m(\overline{x})[P, Q]$ when the body does not lead to suspension nor self calls of methods redefined below B. Otherwise, new pre/post conditions for a method body can be established by new analysis of the body. Thus $\vdash_C B : m(\overline{x})[P, Q]$ follows from $\vdash_B B : m(\overline{x})[P, Q]$ and $\mathit{body}_{C::B:m} = \mathit{body}_{B::B:m}$. The latter condition can be guaranteed by static analysis considering all possible self calls, which gives an integration of Hoare logic and static checking.

Verification of the Example

Let B denote BANK and let I_B denote the invariant of B. From our definition of class verification we get the following verification conditions for class BANK

1. $\vdash_B I_B \Rightarrow I_{PerfectBank}(\mathbf{h}/\mathit{PerfectBank})$ (entailment of interface invariant)
2. $\vdash_B I_B \,{}^{\mathbf{h},\mathit{bal}}_{\mathit{empty},0}$ (class initialization establishes I_B)
3. $\vdash_B \mathit{bal}(x)[I_B, I_B]$ (implementation of *bal* maintains I_B)
4. $\vdash_B \mathit{add}(x)[I_B, I_B]$ (implementation of *add* maintains I_B)
5. $\vdash_B \mathit{sub}(x)[I_B, I_B]$ (implementation of *sub* maintains I_B)
6. $\vdash_B \mathit{bal}(x)[I_B, \mathsf{return} = \mathit{sum}(\mathbf{h}/\mathit{Bank})]$ (pre/post spec. from *Bank*)
7. $\vdash_B \mathit{bal}(x)[\mathit{true}, \mathsf{return} = \mathit{bal}]$ (pre/post spec. from B)
8. $\vdash_B \mathit{add}(x)[\mathit{true}, \mathsf{return} = \mathit{true}]$ (pre/post spec. from B)
9. $\vdash_B \mathit{sub}(x)[\mathit{true}, \mathsf{return} = \mathit{true}]$ (pre/post spec. from B)
10. $\vdash_B \mathit{upd}(x)[\mathit{true}, \mathsf{return} = \mathit{true}]$ (pre/post spec. from B)

11. $\vdash_B upd(x)[I_B, bal = sum(\mathbf{h}) + x \wedge \mathsf{return} = true]$ (pre/post spec. from B)

Here (1) is trivial since there is no *PerfectBank* invariant. Verification conditions (2-5) state that I_B is an invariant of B, and (6-11) ensure the stated pre/post specifications. Note that (8) ensures the pre/post specification for *add* from *Bank*, and (9) ensures the one for *sub* from *PerfectBank*. The verification of the conditions is straight forward as it does not involve any superclass code. For (6) the class invariant in the precondition is needed. History projections may be ignored in this example due to the others clause in the definition of *sum*. Note that the private method *upd* does not maintain the invariant.

According to the definition of class correctness, verification of class BANKPLUS amounts to the following conditions (letting BP denote BANKPLUS)

1. $\vdash_{BP} I_{BP} \Rightarrow sum(\mathbf{h}/BankPlus) \geq 0$ (implication of invariants)
2. $\vdash_{BP} I_{BP}\,{}^{\mathbf{h},bal}_{empty,0}$ (establishment of BP inv.)
3. $\vdash_{BP} B : bal(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.)
4. $\vdash_{BP} B : add(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.)
5. $\vdash_{BP} B : sub(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.)
6. $\vdash_{BP} B : bal(x)[I_{BP}, \mathsf{return} = sum(\mathbf{h})]$ (pre/post given in *Bank*)
7. $\vdash_{BP} B : bal(x)[true, \mathsf{return} = bal]$ (pre/post given in B)
8. $\vdash_{BP} B : add(x)[true, \mathsf{return} = true]$ (pre/post given in B)
9. $\vdash_{BP} B : sub(x)[b' = bal, \mathsf{return} = b' \geq x]$ (pre/post given in B)
10. $\vdash_{BP} upd(x)[b' = bal, \mathsf{return} = b' + x \geq 0]$ (pre/post given in BP)
11. $\vdash_{BP} upd(x)[I_{BP}, bal \geq 0 \wedge bal = sum(\mathbf{h}) + \mathbf{if}\ \mathsf{return}\ \mathbf{then}\ x\ \mathbf{else}\ 0]$ (BP)

Verification of these conditions can be summarized as follows: (1,2,3) are trivial, (4,5) follow from (11), (6) follows from (3,7) by entailment, (8,9) follow from (10), (7) follows from verification of BANK by observing that $body_{BP::B:bal}$ equals $body_{B::B:bal}$ since the body has no self calls nor suspension, and (10,11) are straight forward, using $body_{BP::B:upd} = body_{B::B:upd}$.

Again we notice that the private method *upd* does not maintain the invariant (and is not required to), that the invariant is needed in the proof of (6), and that significant reuse of proofs from the verification of BANK was possible (7-11).

4 Related Work

Class inheritance is a central feature of object orientation which allows subclasses to be designed by reusing and redefining the code of superclasses with a flexibility that goes beyond behavioral subtyping [26, 5]. However, proof systems usually restrict code reuse to behavioral subtyping [16, 3]. For example, a recent survey of challenges and results for the verification of sequential object-oriented programs [14] relies on behavioral subtyping when reasoning about late binding and inheritance. In contrast, proof systems studying late bound methods without relying on behavioral subtyping have been shown to be sound and

complete by Pierik and de Boer [22], but assume a closed world (i.e., all classes must be known).

Problems related to object patters have been studied recently. Designs with collaborating objects pose problems with respect to modular reasoning about common (non-local) invariants. [28] and [17] discuss reasoning when such invariants may be temporarily broken. [28] controls reasoning about invariants that are broken at certain points in the program. [17] provides specific constructs controlling the invariants. A notion of global invariants for collaborating objects is suggested in [24] considering sequential OO programs and patterns. For instance in the observer/subject pattern, the invariant for the pattern is placed in the observer class and its verification involves both classes. In our system, the invariant would partly be in the subject class (say expressing that all subscribers have been notified about the current "value") and partly in the observer class (expressing that the local copy of the "value" has been updated according to the notifications). Interfaces are not used, but the paper mentions reverification of inherited code as a way to achieve better flexibility in inheritance. Another typical example of non-local invariants is the handling of doubly-linked lists. In our setting doubly-linked lists can be treated by ensuring that when a "previous" pointer is set, the "next" pointer of the previous object is correct. This can be expressed as a local property by means of an invariant saying that the *setnext* call has ended before the matching *setprevious* call has ended.

Specification inheritance is used to enforce behavioral subtyping in [8], where subtypes inherit specifications from their supertypes. Virtual methods [23] similarly allow incremental reasoning by committing to certain abstract properties about a method, which must hold for all its implementations. In particular, the verification platforms for *Spec#*[4] and JML [5] rely on versions of behavioral subtyping. Wehrheim [29] investigates behavioral subtyping for active, concurrent objects, classifying different notions of behavioral sybtyping.

The fragile base class problem emerges when seemingly harmless superclass updates lead to unexpected behavior of subclass instances [18]. Many variations of the problem relate to imprecise specifications and assumptions made in super- or subclasses. By supporting static method calls one can refer to and reuse original versions of methods, making method requirements and assumptions explicit, which reduces the fragile base class problem.

Recently incremental reasoning, both for single and multiple inheritance, has been considered in the context of *separation logic* [6, 21]. These approaches distinguish "static" specifications, given for each method implementation, from "dynamic" specifications used to verify late-bound calls, somewhat similar to the approach of lazy behavioral subtyping [11], discussed earlier.

In order to obtain a more flexible specification language, it has been suggested to allow clauses like: this **is** $C \Rightarrow I$, which expresses that the clause only needs to hold when the current object is exactly a C object (and not a subclass object) [14]. This allows more complete invariant (or pre/post) specification of a particular class without imposing such clauses on subclasses, since this **is** C is false in a (proper) subclass of C. However, for reasoning about remote calls with behavioral subtyping, the restrictions of behavioral subtyping remain.

When reasoning with class invariants, the framework should determine in which states the invariant should hold (so-called *invariant states*). Clearly the set of invariant states should be as small as possible without compromising soundness, since this allows stronger invariants. Callbacks should only appear

in invariant states. Then any external call may assume the invariant. In our approach the class invariant I_C must hold in any state where suspension occurs or where an externally called method is completed. This allows us to assume the invariant after suspension. Therefore public methods must maintain the class invariant. Blocking self calls to private or public methods need not be done in invariant states; however, such calls may lead to suspension, in which case one would need to prove $\vdash_C [P]\ s\ [I_C]$ where s is the path to the suspension point and P is the condition at the point of the self call. And if the suspension is a suspending call to a private method, one needs to verify that the method maintains the invariant. The current analysis is able to detect this, as well as the relevant class context, due to the $\vdash_C$ notation. Thus in our system class verification guarantees that callbacks happen in invariant states. There is no need for pack/unpack operation to control class invariants [15].

5 Conclusions

We have presented a verification logic for modular reasoning of concurrent object-oriented programs supporting free reuse of code and specifications. In contrast to earlier approaches, the reasoning of a class does not impose restrictions on subclasses. Object variables are typed by interfaces and remote field access is not supported. Each class C is verified separately, and reused superclass code is re-analyzed under the assumption that this object is of class C. We use history-based specifications, which allow compositional reasoning. The main complication of our logic concerns reasoning about self calls and reused superclass code, something which was not worked out in [19]. Our solution uses a notion of proof context and a notation for static method binding. Our framework considers all main aspects of object-orientation, and represents a general solution that may be used for other object-oriented languages with late binding and inheritance, including sequential languages, but assumes typing with interfaces and no remote field access.

The mechanism of static calls proved helpful for reuse of pre/postconditions for superclass methods. It is also helpful in controlling the fragile base class problem [18]. The considered language involves some additional challenges caused by concurrency with suspension mechanisms and non-blocking calls. Nonetheless, our reasoning system gives rise to quite simple Hoare-style reasoning, similar to reasoning about sequential programs, with the addition of sequential effects on the local history. Our notion of inheritance is flexible with respect to reuse of both code and specifications (more flexible than [19]). The distinction of public and private methods was essential in practice, as demonstrated in the example.

Apart from the (non-trivial) formulation of specifications, our system gives rise to automatic generation of verification conditions, where left-constructive Hoare analysis gives a verification condition at each suspension point and method start. For verification of a method pre/post pair one may first check if it follows from earlier results by the entailment rule, and if not, analyze the body. In future work, we would like to build an automated verification system based on our approach, for instance using the KeY system, which already has support for a version of our language without inheritance [2].

The integration aspect of this work lies in the combination of a Hoare logic and an equivalence over inherited and non-inherited code, which allows reuse of

verification results in subclasses. The example illustrates the value of verification reuse, showing that all cases of reuse of a method and its specification resulted in verification reuse. The equivalence of code (i.e., of method bodies) is detected by static analysis and exploited in verification. In addition, the static detection of $o \neq \mathsf{this}$ gives simplification at the reasoning level.

In the reasoning system we have not considered loops and recursion, which can be handled as usual. Extension to multiple inheritance should be possible (for instance solving the diamond problem as in [11]). A discussion of soundness is beyond the scope of this paper. A main part of the soundness proof would be to establish that the class context reflects the run-time class of the executing object. Soundness for the language without inheritance can be done as in [10].

Acknowledgment The anonymous referees have provided valuable feedback.

References

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

[2] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309, 2012.

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

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. et.al, editor, *Intl. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[5] 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.

[6] W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In POPL'08 [1], pages 87–99.

[7] O.-J. Dahl. *Verifiable Programming.* International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[8] 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.

[9] C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27:1–22, 2014.

[10] C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(5-6):360–383, 2014.

[11] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

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

[13] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, 2006.

[14] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.*, 37(4):13, 2015.

[15] K. R. M. Leino and A. Wallenburg. Class-local object invariants. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 57–66, New York, NY, USA, 2008. ACM.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. on Progr. Lang. and Syst.*, 16(6):1811–1841, Nov. 1994.

[17] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electron. Notes Theor. Comput. Sci.*, 195:211–229, Jan. 2008.

[18] 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 *LNCS*, pages 355–382. Springer, 1998.

[19] O. Owe. Verifiable programming of object-oriented and distributed systems. In L. Petre and E. Sekerinski, editors, *From Action System to Distributed Systems: The Refinement Approach*, pages 61–80. Taylor and Francis, 2015.

[20] O. Owe and I. Ryl. On combining object orientation, openness and reliability. In *Proc. of the Norwegian Informatics Conf. (NIK'99)*. Tapir, Nov. 1999.

[21] M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In POPL'08 [1].

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

[23] 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 *LNCS*, pages 162–176. Springer, 1999.

[24] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. *CoRR*, abs/1311.6329, 2013.

[25] M. Pradel and T. R. Gross. Automatic testing of sequential and concurrent substitutability. In *International Conference on Software Engineering (ICSE)*, 2013.

[26] 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.

[27] N. Soundararajan. A proof technique for parallel programs. Theoretical Computer Science, 31(1-2):13–29, May 1984.

[28] A. J. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 328–344, Berlin, Heidelberg, 2010. Springer-Verlag.

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