A Framework for Reasoning about Object-Oriented Concurrent Systems

Olaf Owe

Department of Informatics, University of Oslo
email: olaf@ifi.uio.no

Dec. 15, 2015
SINTEF

Why program reasoning?

Verifiable programming

- so that program verification is as easy as possible
- also automatically!

A program that is easy to verify is

- easy to understand
- also by others
- well specified
- modular

OJD: programmers should know about program verification even if they do not do it.

How?

Program development and program *correctness* hand-in-hand

- reasoning early: at modeling time
- executable modeling:
  - may simulate, test, and verify program components
- imperative, executable modeling such that correctness is relevant to the final program

Reasoning

Program verification depends on

- programming language and its semantics
- specification language
- reasoning system

We aim at simplicity and mechanizability in reasoning.

Several dimensions of added complexity:

- concurrency
- object-orientation (OO)
- program changes/maintenance
- security
- ...

Object-Orientation

complications

- programming language semantics
  - inheritance
  - unrestricted reuse
  - visibility
  - static/dynamic binding
- specification language
  - abstract interface specifications
  - invariant specifications
  - method specifications
- reasoning system
  - inheritance, reuse, and late binding
  - dynamic generation
  - compositional reasoning

and also inheritance anomalies and fragile base class problem.

Reasoning about Object-Orientation

- programming language semantics: simplicity
  - visibility: by interfaces
- specification language: *local communication history* ***h***
  - use abstract interface specifications over **h**: interface invariant
  - class invariant specifications over **h** and fields
  - method specifications over **h**, fields and parameters
- reasoning system
  - reason locally about each class, modularity
  - allow inheritance, unrestricted reuse, and late binding
  - dynamic generation by communication history
  - compositional reasoning, forming a global invariant over the global history by conjunction of local histories

and also inheritance anomalies and fragile base class problem:
Static method calls (to superclasses) + class-specific specification.
Allow static method calls to fix the binding of super-class methods.

Concurrency challenges

- programming language
  - what can be parallelized
  - synchronization
  - waiting, notification, locks
  - open/dynamic/changing environments
- specification language
  - how to specify cooperation and interaction
  - how to specify local actions
  - visibility
- reasoning system
  - compositional reasoning, class-wise
  - local reasoning
  - global reasoning
  - interference

Reasoning about Concurrency challenges

- programming language
  - what can be parallelized: objects
  - avoid explicit synchronization
  - waiting, notification, locks: avoid
  - open/dynamic/changing environments
- specification language
  - how to specify cooperation and interaction: **h**
  - how to specify local actions: pre/post spec
  - visibility: interface invariant using Red**h**
- reasoning system
  - compositional reasoning, class-wise: invariants
  - local reasoning: class invariants
  - global reasoning: composition rule
  - interference: none

Program Evolution and Maintenance

Important aspects of program development:

- reuse of code
- reuse of specifications
- reuse of proofs

Important aspects of program maintenance:

- changing/updating code
- changing/updating specifications
- redoing/updating proofs

Outline

We give a framework, consisting of

- a core programming language
- a specification language
- a reasoning system

We illustrate the approach by

- a small (1 page) example
  - illustrating the problems of unrestricted reuse
- a core language with a simple semantics
  - for *concurrent objects*
  - communicating by *asynchronous method calls*,
  - with *suspension* to avoid busy-waiting.
- a Hoare-style logic

The language is a revision of *Creol*,

- more high-level (future-free) and
- with revised specification mechanisms and semantics.

The Considered Language

- each object has its own (virtual) processor
- remote methods call implemented by two-way message passing
- conditional suspension **await** *condition* – passive waiting

Dynamic calls:

- called with dot-notation, possibly self-calls using this
- simple call $o.m(\overline{e})$ – no result needed, no waiting
- blocking call $result := o.m(\overline{e})$, or local call $result :=$ this$.m(\overline{e})$
- non-blocking call **await** $result := o.m(\overline{e})$

(Assuming the interface of o offers a type-correct method m.)

Static calls:

- $result := B : m(\overline{e})$, taking the m of (super)class B

Syntax (with specification elements in blue)

Pr	::=	$[In^*\ Cl]^+$	program
In	::=	**interface** $F([T\ p]^*)$	
		$[$**extends** $[F(\overline{e})]^+]^?\{S^*\ I\}$	interface definition
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	::=	$F \mid$ Any $\mid$ Void $\mid$ Bool $\mid$ String $\mid$ Int $\mid$ Nat $\mid \ldots$	types (Note: not C)
v	::=	$x \mid w$	variables, local/field
e	::=	null $\mid$ *void* $\mid$ this $\mid$ caller $\mid v \mid cp \mid f(\overline{e}) \mid (e)$	pure expressions
s	::=	**skip** $\mid v := e \mid v :=$ **new** $C(\overline{e}) \mid s; s$	basic statements
		$\mid [v :=]^? e.m(\overline{e}) \mid [v :=]^? C : m(\overline{e})$	call statements
		$\mid$ **await** $v := e.m(\overline{e}) \mid$ **await** e	suspending statem.
		$\mid$ **if** e **then** s [**else** $s]^?$ **fi**	if statement
P	::=	**[** A, A **]**	pre/postconditions
I	::=	$[$**inv** $A]^?$ $[$**where** $A^+]^?$	invariant

Specifications

- class/interface invariants
- subclasses inherit "everything not touched"
- interfaces inherit everything
- pre/postcondition of methods in interfaces/classes
- may talk about the *local history* **h**
- (framing, but not needed here).

May freely override specifications and code!

Inheritance of Code and Specifications

We let **`inherits`** mean "inherit everything you do not touch", i.e.:

- inherit a method body unless redefined
- inherit a class initialization, unless redefined
- inherit all method pre/post specifications of a method unless new pre/post specifications are given
- inherit a class invariant unless another is given
- inherit an **`implements`** clause unless another is given
- inherit all auxiliary functions (defined after **`where`**)

implements *I*: the class must respect all *I* specifications.

A Simple Bank Example: Interfaces

```
interface Bank {
  Bool sub(Nat x)
  Bool add(Nat x) [true, return=true]
  Int bal() [true, return=sum(h)]
  where sum(empty) = 0,
    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) [true, return=true]}

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

Note: In postconditions, **return** denotes the returned value. The specification function *sum* is defined inductively over the local history **h**, see below:

History-Based Specifications

The local history **h** of a class/interface is the time sequence of communications events seen by this object:

$\text{this} \rightarrow o.m(\overline{e})$ — a method call made by this object,

$o \twoheadrightarrow \text{this}.m(\overline{e})$ — a method call received by this object,

$o \leftarrow \text{this}.m(\overline{e}; e)$ — a method return made by this object,

$\text{this} \twoheadleftarrow o.m(\overline{e}; e)$ — a method return received by this object,

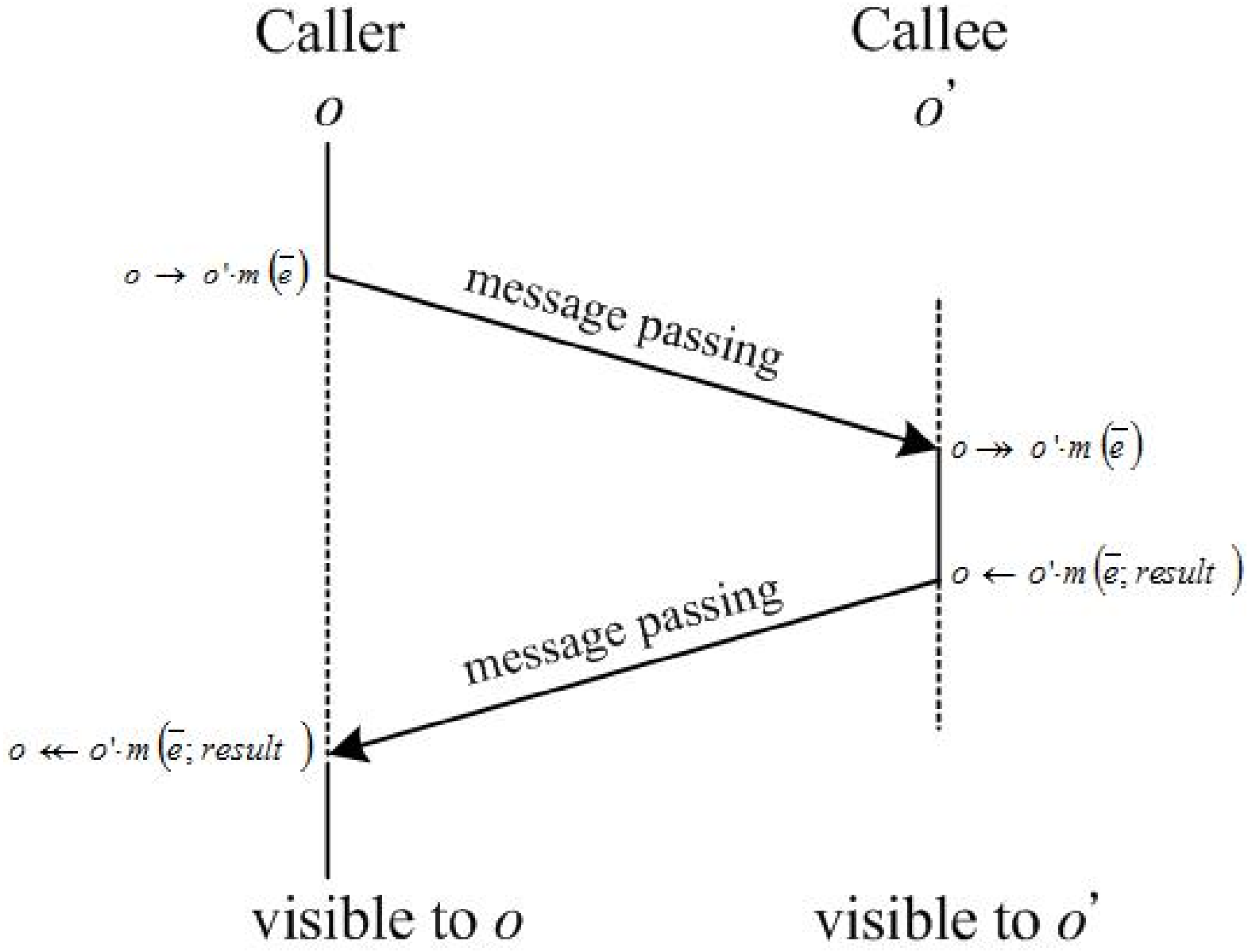
$\text{this} \rightarrow o.\,\mathbf{new}\, C(\overline{e})$ — a creation events made by this object.

Note: Different objects have disjoint event alphabets!
Leads to simple compositional reasoning.

Notation:

$\mathbf{h}; e$ denotes **h** appended with the event e

A call cycle: o calling $o'.m(\overline{e})$



A simple Bank example: Classes

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

Note: In specifications, **inv** refers to the current invariant, and **C:inv** refers to the invariant of C.

A simple Bank example: a possible client

```
class CLIENT {
  Seq[String] paid; Bank acc;
  acc:= new BANK; -- initialization

  Bool salary(Nat x){Bool ok; ok := acc.add(x); return ok} -- blocking

  Bool bill(String kid, Nat x, Bank y){Bool ok:=false;
    if not kid in paid then await ok := acc.sub(x); -- non-blocking
      if ok then y.add(x); paid:=(paid;kid) fi fi;
    return ok }

  inv paid=allpaid(h) -- paid reflects successful payments
  where
    allpaid(empty) = empty,
    allpaid(h; (_←this.bill(k,x,y;true))) = (allpaid(h); k),
    allpaid(h; others) = allpaid(h)
}
```

Note: illustrating the other call mechanisms.

The Example

In the example

- BANKPLUS violates the restrictions of behavioral subtyping: BANKPLUS does not respect interface PerfectBank, and not the pre/post specification of *sub* from BANK.
- BANKPLUS violates restrictions of lazy behavioral subtyping: since BANKPLUS violates the pre/post specifications of *upd* needed in BANK.

Note: Our approach handles the reuse in this example, and the given specifications are sufficient. We define a *Hoare-style logic*:

Hoare style reasoning

For type-correct programs, our language gives simple reasoning:

- assignment rule
- sequential composition
- implication/adaptation/entailment rules

The local history **h** involved in rules for

- call
- object generation
- suspension

Assignment Axiom

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

- Backwards reasoning!
- Example: Find precondition P such that $[P]\, x := x + y\, [x = y]$. The rule gives $x + y = y$, i.e., $x = 0$.
- Aliasing is a problem: $[x = y]\, x.v := x.v + y.v\, [x = y]$ does not hold, but should hold if x and y are aliases.

We allow aliasing, but not remote access to fields, no shared variables. *Then assignment axiom is sound!*

Sequential Composition

$$\frac{\vdash [P]\ S1\ [Q] \qquad \vdash [Q]\ S2\ [R]}{\vdash [P]\ S1; S2\ [R]}$$

- Concurrency: Interference?
- For each object, the local conditions are on disjoint variables. Histories of two objects do not share events!

Implication rule

Implication and conjunction rules as usual.

$$\frac{\vdash P \Rightarrow P',\ \vdash [P']\ S2\ [Q'],\ \vdash Q \Rightarrow Q}{\vdash [P]\ S1\ [Q]}$$

However, we may have severel pre/post pairs for each method, and may need to derive facts from all of them.

A multi pre/post specification example

Given:

$$sub(x)[x \leq bal, return = true]$$

$$sub(x)[x > bal, return = false]$$

How can we derive

$$sub(x)[b0 = bal, return = (x \leq b0)]$$

We need a generalized rules, able to derive facts from several pre/post pairs.

Note: Here $m(x)[P, Q]$ means $[P]\ body_m\ [Q]$.

Entailment

Generalized implication/adaptation rule:

$$\vdash_C B : m(\overline{x})[P_j, Q_j], \text{all } j \in J$$

$$\frac{\vdash_C \ (\wedge_{j \in J}[[P_j, Q_j]]) \ \Rightarrow \ [[P, Q]]}{\vdash_C B : m(\overline{x})[P, Q]}$$

where $[P, Q]$ is the relational meaning of the pre/post condition pair (given as an input-ouput relation, using primes for post-values):

$$[[P, Q]] = \forall z \,.\, P \Rightarrow Q^{w,\mathbf{h}}_{w',\mathbf{h}'}$$

where z is the list of logical variables in $[P, Q]$, w is the list of fields of C and and w' are fresh logical variables (representing post-state values).

Inter-Object Interaction

- simple call (without waiting for the result)

$$\vdash [Q^{\mathbf{h}}_{\mathbf{h};(\mathsf{this}\rightarrow o.m(\overline{e}))}]\ o.m(\overline{e})\ [Q]$$

- blocking call (wait for the result)

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

Note: "this" is blocked. Nothing happens to its local history!

- object generation

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

Note: straight forward, due to histories and interface abstraction.

Main Idea for Local Calls and Late Binding

The execution of a method body given in a class B depends on the class of this object, say C, which is used for binding of calls inside the body. We use the notation

$$body_{C::B:m}$$

- A dynamic call this.$m(\overline{e})$ binds to $body_{C::B:m}$ if C is the class of this and B the closest superclass of C with a definition of m
- A static call $B:m(\overline{e})$ binds to $body_{C::B:m}$ where C is the class of this.

Main Idea: For each class C, analyze all exported methods m (defined/redefined/inherited) using $body_{C::C:m}$
and ensure that the interface(s) of C are satisfied.

Note: Then the run-time class of "this" is statically known.

Use $\vdash_C [P]\, s\, [Q]$ rather than $\vdash [P]\, s\, [Q]$ to fix the class of "this".

Local Calls and Self Calls

- self call reduces to static local call

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

- static local calls

$$\frac{\vdash_C [P]\ body_{C::B:m}\ [Q^{\mathbf{h}}_{\mathbf{h};(\mathsf{this}\twoheadleftarrow\mathsf{this}.m(\overline{e};v))}]}{\vdash_C [P^{\overline{x},\mathsf{caller},\mathbf{h}}_{\overline{e},\mathsf{this},\mathbf{h};(\mathsf{this}\rightarrow\mathsf{this}.m(\overline{e}))} \wedge L]\ v := B : m(\overline{e})\ [Q^{\overline{x},\mathsf{caller},\mathsf{return}}_{\overline{e},\mathsf{this},v} \wedge L]}$$

where L is a condition without fields modified by $body_{C::B:m}$, and where $body_{C::B:m}$ is

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

given that B has the method definition $m(\overline{x})\{s;\ \mathbf{return}\ e\}$.
(We assume here v does not occur in e.)

Reasoning about history and suspension

- The history grows with time

$$\vdash_C [h' = \mathbf{h}]\, s\, [h' \leq \mathbf{h}]$$

- Suspension relies on the invariant:

$$\vdash_C [I_C \wedge L]\ \mathtt{await}\ b\, [b \wedge I_C \wedge L]$$

where

- I_C is the invariant of C, and
- L is a "local condition", i.e., not referring to fields of C.

Note: Need to be aware of the context class C (rather than just the enclosing class). Suspension and L are sensitive to C!

$\mathrm{body}_{\mathrm{C::B:m}}$ equals $\mathrm{body}_{\mathrm{B::B:m}}$ when no local calls nor suspension.

Verification of a class C

- $\vdash_C I_C \Rightarrow I_F(\mathbf{h}/F)$, for each invariant I_F of an interface F of C
- $\vdash_C init()[true, I_C]$, the class initialization establishes I_C
- $\vdash_C m(\overline{x})[I_C, I_C]$, for each C-public method m
- $\vdash_C m(\overline{x})[I_C \wedge P, Q]$, if F contains $m(\overline{x})[P, Q]$
- $\vdash_C m(\overline{x})[P, Q]$, if C contains $m(\overline{x})[P, Q]$

where

- $\vdash_C m(\overline{x})[P, Q]$ denotes $\vdash_C [P]\ body_{C::C:m}\ [Q]$
- $\vdash_C B : m(\overline{x})[P, Q]$ denotes $\vdash_C [P]\ body_{C::B:m}\ [Q]$
- $\mathbf{h}/F$ is the projection of $\mathbf{h}$ to the alphabet of F
- C contains means in C or inherited in C.

Verification of the example: class BANKPLUS

Let B denote BANK and BP denote BANKPLUS.

$\vdash_{BP} (I_{BP} \Rightarrow sum(\mathbf{h}/BankPlus) \geq 0)$ (implication of invariants) (1)

$\vdash_{BP} I_{BP}{}^{\mathbf{h},bal}_{empty,0}$ (establishment of BP inv.) (2)

$\vdash_{BP} B : bal(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.) (3)

$\vdash_{BP} B : add(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.) (4)

$\vdash_{BP} B : sub(x)[I_{BP}, I_{BP}]$ (maintenance of BP inv.) (5)

$\vdash_{BP} B : bal(x)[true, \mathsf{return} = bal]$ (given pre/post) (6)

$\vdash_{BP} B : add(x)[true, \mathsf{return} = true]$ (given pre/post) (7)

$\vdash_{BP} upd(x)[I_{BP}, bal \geq 0 \wedge bal = sum(\mathbf{h}) + \mathbf{if}\ \mathsf{return}\ \mathbf{then}\ x\ \mathbf{else}\ 0]$ (8)

$\vdash_{BP} upd(x)[b' = bal, \mathsf{return} = (b' + x \geq 0)]$ (given pre/post) (9)

- (1,2,3) are trivial.
- (6) follows from verification of BANK by observing that $body_{BP::B:bal}$ equals $body_{B::B:bal}$ since no local calls/suspension.
- (4,5) follow from (8), and (7) from (9).
- (8,9) are straight forward, using $body_{BP::B:upd} = body_{B::B:upd}$.

Summary Example

- essential that not all methods must maintain the class invariant
- pubic /private controlled by the interfaces
- reuse of proof due to "body equality"
- free reuse was needed in the example
- flexible specification inheritance needed
- mullti-specification (entailment) was needed
- class invariant needed when verifying interface preconditions

All verification conditions small, and mechanizable!

Suitable for automation!

Contributions

- sequential like reasoning
- concurrency "for free"
- reasoning for flexible reuse with behavioral interface subtyping
- reasoning about inheritance and late binding
- + static calls – fragile base class problem

  - flexible and simple specification inheritance
  - worked out for concurrent object, async. methods, suspension
  - quite simple Hoare-style reasoning, like sequential reasoning apart from effects on the history

Based on interface abstraction (typing with interfaces)

Simple semantics gives simple reasoning

Generality

Generality, may use the same approach on

- software changes on classes, without affecting superclasses
- other core languages, but possibly more complex reasoning
- other concurrency models, or sequential OO languages
- For Java: interfaces for hiding, and variable declarations, threads -> active concurrent objects, no remote field access, no locks/sync./notify/wait
- more flexible typing with both interfaces and classes, using (lazy) behavioral subtyping for such classes.

References

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

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

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.

L. Mikhajlov and E. Sekerinski.
A study of the fragile base class problem.
In *ECOOP'98 Object-Oriented Programming*, pages 355–382. Springer, 1998.

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&Francis, 2015.