Assertion-based encapsulation and refinement of classes

David A. Naumann

Stevens Institute of Technology

Joint work with Mike Barnett and Anindya Banerjee

Supported by NSF CCR-0208984 and CCF-0429894.

Outline

- Difficulty in reasoning about object invariants due to callbacks and heap sharing (in sequential programs) —programmer's view, logician's view
- The boogie and friends disciplines: state based encapsulation (with Mike Barnett [LICS])
- Simulations (two-state invariants) —representation independence (with Anindya Banerjee)
- Application of simulation: observational purity
- Related work and Phd/postdoc advert

Programmer's intro: object invariants

```
class Subject {
   private x,y: int := 0,1;
   invariant 𝓘(self)    where 𝓘 is defined by 𝓘(o) = o.x<o.y
   method m() {
      self.x := self.x+1;

      self.y := self.y+1; }    ... }
```

Programmer's intro: object invariants

```
class Subject {
   private x,y: int := 0,1; obs: Observer :=. . . ;
   invariant 𝓘(self)    where 𝓘 is defined by 𝓘(o) = o.x<o.y
   method m() {
      self.x := self.x+1;
      obs.notify();
      self.y := self.y+1; }    . . . }
```

Programmer's intro: object invariants

```
class Subject {
   private x,y: int := 0,1; obs: Observer :=. . . ;
   invariant 𝓘(self)    where 𝓘 is defined by 𝓘(o) = o.x<o.y
   method m() {
      self.x := self.x+1;
      obs.notify();
      self.y := self.y+1; }    . . . }
class Observer {
   z: Subject := . . . ;
   method notify() { z.m(); }    . . . }
```

Programmer's intro: object invariants

```
class Subject {
   private x,y: int := 0,1; obs: Observer :=. . . ;
   invariant 𝓘(self)    where 𝓘 is defined by 𝓘(o) = o.x<o.y
   method m() {
      self.x := self.x+1;
      obs.notify();
      self.y := self.y+1; }    . . . }
class Observer {
   z: Subject := . . . ;
   method notify() { z.m(); }    . . . }
```

When should $\mathcal{I}$ hold?

Programmer's intro (2): sharing

```
class Subject2 {
   private x: Integer := new Integer(0);
   private y: Integer := new Integer(1);
   invariant 𝓘(self)    where 𝓘(o) = o.x.val<o.y.val
   method m() { self.x.incr(); self.y.incr(); }
                                                    }
```

Programmer's intro (2): sharing

```
class Subject2 {
   private x: Integer := new Integer(0);
   private y: Integer := new Integer(1);
   invariant 𝓘(self)    where 𝓘(o) = o.x.val<o.y.val
   method m() { self.x.incr(); self.y.incr(); }
   method leak(): Integer { result := x; }     }
class Main {
   s: Subject2; i: Integer;
   . . . i := s.leak(); i.incr(); s.m() . . . }
```

Programmer's intro (2): sharing

```
class Subject2 {
   private x: Integer := new Integer(0);
   private y: Integer := new Integer(1);
   invariant 𝓘(self)    where 𝓘(o) = o.x.val<o.y.val
   method m() { self.x.incr(); self.y.incr(); }
   method leak(): Integer { result := x; }     }
class Main {
   s: Subject2; i: Integer;
   . . . i := s.leak(); i.incr(); s.m() . . . }
```

How can we encapsulate not just fields but also referenced objects?

Logician's intro

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathtt{body}\ \{\mathcal{Q} \wedge \mathcal{I}\} \qquad \mathcal{I} \text{ is encapsulated for } \mathtt{m}}{\{\mathcal{P}\}\ \textbf{call}\ \mathtt{m}()\ \{\mathcal{Q}\}}$$

Declaration: $\mathtt{m}()\{\mathtt{body}\}$. *Specification* $\mathcal{P}, \mathcal{Q}$ *involves public fields;* $\mathcal{I}$ *depends only on the internal representation.*

Logician's intro

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathtt{body}\ \{\mathcal{Q} \wedge \mathcal{I}\} \qquad \mathcal{I} \text{ is encapsulated for } \mathtt{m}}{\{\mathcal{P}\}\ \textbf{call}\ \mathtt{m}()\ \{\mathcal{Q}\}}$$

Declaration: $\mathtt{m}()\{\mathtt{body}\}$. *Specification* $\mathcal{P}, \mathcal{Q}$ *involves public fields;* $\mathcal{I}$ *depends only on the internal representation.*

$$\frac{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\} \qquad S \text{ does not interfere* with } \mathcal{I}}{\{\mathcal{P} \wedge \mathcal{I}\}\ S\ \{\mathcal{Q} \wedge \mathcal{I}\}}$$

Logician's intro

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathtt{body}\ \{\mathcal{Q} \wedge \mathcal{I}\} \qquad \mathcal{I} \text{ is encapsulated for } \mathtt{m}}{\{\mathcal{P}\}\ \textbf{call}\ \mathtt{m}()\ \{\mathcal{Q}\}}$$

Declaration: $\mathtt{m}()\{\mathtt{body}\}$. *Specification* $\mathcal{P}, \mathcal{Q}$ *involves public fields;* $\mathcal{I}$ *depends only on the internal representation.*

$$\frac{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\} \qquad S \text{ does not interfere* with } \mathcal{I}}{\{\mathcal{P} \wedge \mathcal{I}\}\ S\ \{\mathcal{Q} \wedge \mathcal{I}\}}$$

* S *does not write variables read in* $\mathcal{I}$ *(hazard: aliased vars)*

* S *does not update objects read in* $\mathcal{I}$ *(hazard: heap sharing)*

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\quad\}\, x := E\, \{\mathcal{P}\quad\}}{\{\mathcal{R}\quad\}\, x := E\, ;\ \textbf{call}\ m\, \{\mathcal{Q}\quad\}} \qquad \dfrac{\dfrac{\vdots}{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathrm{body}\ \{\mathcal{Q} \wedge \mathcal{I}\}}}{\{\mathcal{P}\quad\}\ \textbf{call}\ m\ \{\mathcal{Q}\quad\}}}{\vdots}$$

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\} \quad \text{no interference}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\wedge\mathcal{I}\}} \qquad \dfrac{\vdots}{\dfrac{\{\mathcal{P}\wedge\mathcal{I}\}\ \mathrm{body}\ \{\mathcal{Q}\wedge\mathcal{I}\}}{\{\mathcal{P}\wedge\mathcal{I}\}\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ ;\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}$$

$$\vdots$$

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\} \quad \text{no interference}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\wedge\mathcal{I}\}} \qquad \dfrac{\dfrac{}{\vdots}}{\dfrac{\{\mathcal{P}\wedge\mathcal{I}\}\ \mathrm{body}\ \{\mathcal{Q}\wedge\mathcal{I}\}}{\{\mathcal{P}\wedge\mathcal{I}\}\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ ;\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}$$

$$\vdots$$

$$\{\mathcal{R}\}\ \mathrm{init};\ \{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ ;\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\} \Rightarrow \{\mathcal{Q}\}$$

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\} \quad \text{no interference}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ \{\mathcal{P}\wedge\mathcal{I}\}} \qquad \dfrac{\vdots}{\dfrac{\{\mathcal{P}\wedge\mathcal{I}\}\ \mathrm{body}\ \{\mathcal{Q}\wedge\mathcal{I}\}}{\{\mathcal{P}\wedge\mathcal{I}\}\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}}}{\{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ ;\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\}}$$

$$\vdots$$

$$\{\mathcal{R}\}\ \mathrm{init};\ \{\mathcal{R}\wedge\mathcal{I}\}\ \mathrm{x} := \mathrm{E}\ ;\ \textbf{call}\ \mathrm{m}\ \{\mathcal{Q}\wedge\mathcal{I}\} \Rightarrow \{\mathcal{Q}\}$$

How generalize to multiple instantiation, i.e., $\mathcal{I}(\mathrm{o})$ *for all* o*?*

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\}\,\mathrm{x} := \mathrm{E}\,\{\mathcal{P}\} \quad \text{no interference}}{\{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,\{\mathcal{P}\wedge\mathcal{I}\}} \qquad \dfrac{\vdots}{\dfrac{\{\mathcal{P}\wedge\mathcal{I}\}\,\mathrm{body}\,\{\mathcal{Q}\wedge\mathcal{I}\}}{\{\mathcal{P}\wedge\mathcal{I}\}\,\textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\}}}}{\{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,;\ \textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\}}$$

$$\vdots$$

$$\{\mathcal{R}\}\,\mathrm{init};\ \{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,;\ \textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\} \Rightarrow \{\mathcal{Q}\}$$

How generalize to multiple instantiation, i.e., $\mathcal{I}(\mathrm{o})$ for all o? What about outcalls in body, i.e. method invocations on other objects, which may lead to reentrant callbacks?

Logicians' intro (2)

$$\dfrac{\dfrac{\{\mathcal{R}\}\,\mathrm{x} := \mathrm{E}\,\{\mathcal{P}\} \quad \text{no interference}}{\{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,\{\mathcal{P}\wedge\mathcal{I}\}} \qquad \dfrac{\vdots}{\dfrac{\{\mathcal{P}\wedge\mathcal{I}\}\,\mathrm{body}\,\{\mathcal{Q}\wedge\mathcal{I}\}}{\{\mathcal{P}\wedge\mathcal{I}\}\,\textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\}}}}{\{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,;\ \textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\}}$$

$$\vdots$$

$$\{\mathcal{R}\}\,\mathrm{init};\ \{\mathcal{R}\wedge\mathcal{I}\}\,\mathrm{x} := \mathrm{E}\,;\ \textbf{call}\ \mathrm{m}\,\{\mathcal{Q}\wedge\mathcal{I}\} \Rightarrow \{\mathcal{Q}\}$$

How generalize to multiple instantiation, i.e., $\mathcal{I}(\mathrm{o})$ for all o?
What about outcalls in body, i.e. method invocations on other objects, which may lead to reentrant callbacks?
How express absence of interference due to heap sharing?

An assertion-based discipline

Problems:

- due to reentrant callbacks, precondition $\mathcal{P} \wedge \mathcal{I}$ for $\mathtt{body}$ is unsound unless $\mathcal{I}$ re-established before outcalls

An assertion-based discipline

Problems:

- due to reentrant callbacks, precondition $\mathcal{P} \wedge \mathcal{I}$ for $\mathtt{body}$ is unsound unless $\mathcal{I}$ re-established before outcalls
- need to protect $\mathcal{I}_{\mathrm{Subject}}(\mathrm{o})$ from interference —by code in other classes and by other instances $\mathrm{p} \neq \mathrm{o}$ of Subject

An assertion-based discipline

Problems:

- due to reentrant callbacks, precondition $\mathcal{P} \wedge \mathcal{I}$ for $\mathtt{body}$ is unsound unless $\mathcal{I}$ re-established before outcalls
- need to protect $\mathcal{I}_{\mathrm{Subject}}(\mathrm{o})$ from interference —by code in other classes and by other instances $\mathrm{p} \neq \mathrm{o}$ of Subject

Solution uses a single everywhere-invariant, $\mathcal{PI}$.

Rule: $\dfrac{\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}}$.

An assertion-based discipline

Problems:

- due to reentrant callbacks, precondition $\mathcal{P} \wedge \mathcal{I}$ for $\mathtt{body}$ is unsound unless $\mathcal{I}$ re-established before outcalls
- need to protect $\mathcal{I}_{\mathrm{Subject}}(\mathrm{o})$ from interference —by code in other classes and by other instances $\mathrm{p} \neq \mathrm{o}$ of Subject

Solution uses a single everywhere-invariant, $\mathcal{PI}$.

Rule: $\dfrac{\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}}$. Handles transfer and sharing of objects across encapsulation boundaries. Can use with standard logics and type systems.

Auxiliary field to make explicit when invariant holds:

inv : **boolean** $:= false$

Maintain program invariant $\mathcal{PI} : (\ \forall\, o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\ )$

Auxiliary field to make explicit when invariant holds:

inv : **boolean** := $false$

Maintain program invariant $\mathcal{PI} : (\ \forall\, o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\ )$

class Subject { . . .

invariant $\mathcal{I}_{Subject}$(self) where $\mathcal{I}_{Subject}$(o) = o.x<o.y

method m() {

assert self.inv (* precondition *)

unpack self; (* self.inv := false *)

self.x := self.x+1; obs.notify(); self.y := self.y+1; (* $\mathcal{I}$(self) *)

pack self; (* self.inv := true *) } . . . }

class Main . . . **method** notify() { **assert** z.inv ?; z.m(); }

Auxiliary field to make explicit when invariant holds:

inv : **boolean** := $false$

Maintain program invariant $\mathcal{PI} : (\forall o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o))$

class Subject { . . .

invariant $\mathcal{I}_{Subject}$(self) where $\mathcal{I}_{Subject}$(o) = o.x<o.y

method m() {

assert self.inv (* precondition *)

unpack self; (* self.inv := false *)

self.x := self.x+1; obs.notify(); self.y := self.y+1; (* $\mathcal{I}$(self) *)

pack self; (* self.inv := true *) } . . . }

class Main . . . **method** notify() { **assert** z.inv ?; z.m(); }

Absence of interf., as a precond.: $\{\neg v.inv \wedge \mathcal{PI}\}\, v.f := E\, \{\mathcal{PI}\}$

Auxiliary field to delimit heap dependence of invariant:

$\mathrm{own} : \mathrm{Object} := \mathrm{null}$

Def: $\mathrm{o} \preceq \mathrm{p}$ *iff either* $\mathrm{o} = \mathrm{p.own}$ *or* $\mathrm{o} \preceq \mathrm{p.own}$.

Auxiliary field to delimit heap dependence of invariant:

$\mathrm{own} : \mathrm{Object} := \mathrm{null}$

Def: $\mathrm{o} \preceq \mathrm{p}$ *iff either* $\mathrm{o} = \mathrm{p.own}$ *or* $\mathrm{o} \preceq \mathrm{p.own}$.

Def: $\mathcal{I}_\mathrm{C}$ *is admissible iff*

when $\mathcal{I}_\mathrm{C}(\mathrm{o})$ *depends on* $\mathrm{p.f}$ *then either* $\mathrm{o} = \mathrm{p}$ *or* $\mathrm{o} \preceq \mathrm{p}$.

Auxiliary field to delimit heap dependence of invariant:

$\mathrm{own} : \mathrm{Object} := \mathrm{null}$

Def: $\mathrm{o} \preceq \mathrm{p}$ *iff either* $\mathrm{o} = \mathrm{p.own}$ *or* $\mathrm{o} \preceq \mathrm{p.own}$.

Def: $\mathcal{I}_C$ *is admissible iff*

when $\mathcal{I}_C(\mathrm{o})$ *depends on* $\mathrm{p.f}$ *then either* $\mathrm{o} = \mathrm{p}$ *or* $\mathrm{o} \preceq \mathrm{p}$.

Absence of interference, as a precondition:

$\{\neg v.\mathrm{inv} \wedge (\,\forall\, \mathrm{o} \bullet \mathrm{o} \preceq v \Rightarrow \neg \mathrm{o.inv}\,) \wedge \mathcal{PI}\}\; v.\mathrm{f} := \mathrm{E}\; \{\mathcal{PI}\}$

Ownership provides stateful encapsulation: $\neg \mathrm{o.inv}$ *means control is inside the boundary for* o.

Last auxiliary field for ownership discipline:

com : **boolean** $:= false$

$$\mathcal{PI}: \ (\forall o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\ ) \wedge$$
$$(\forall o \bullet o.inv \Rightarrow (\forall p \bullet p.own = o \Rightarrow p.com\ )) \wedge$$
$$(\forall o \bullet o.com \Rightarrow o.inv\ )$$

Last auxiliary field for ownership discipline:

$\mathrm{com} : \textbf{boolean} := \mathrm{false}$

$$\mathcal{PI}: \ (\forall \mathrm{o} \bullet \mathrm{o.inv} \Rightarrow \mathcal{I}_{type(\mathrm{o})}(\mathrm{o})\ ) \wedge$$
$$(\forall \mathrm{o} \bullet \mathrm{o.inv} \Rightarrow (\forall \mathrm{p} \bullet \mathrm{p.own} = \mathrm{o} \Rightarrow \mathrm{p.com}\ )) \wedge$$
$$(\forall \mathrm{o} \bullet \mathrm{o.com} \Rightarrow \mathrm{o.inv}\ )$$

Absence of interf., as a precond.: $\{\neg v.\mathrm{inv} \wedge \mathcal{PI}\}\, v.\mathrm{f} := \mathrm{E}\, \{\mathcal{PI}\}$

Last auxiliary field for ownership discipline:

$\mathrm{com} : \textbf{boolean} := \mathit{false}$

$\mathcal{PI}: \ (\forall \mathrm{o} \bullet \mathrm{o.inv} \Rightarrow \mathcal{I}_{type(\mathrm{o})}(\mathrm{o})\ ) \wedge$

$\quad (\forall \mathrm{o} \bullet \mathrm{o.inv} \Rightarrow (\forall \mathrm{p} \bullet \mathrm{p.own} = \mathrm{o} \Rightarrow \mathrm{p.com}\ )) \wedge$

$\quad (\forall \mathrm{o} \bullet \mathrm{o.com} \Rightarrow \mathrm{o.inv}\ )$

Absence of interf., as a precond.: $\{\neg v.\mathrm{inv} \wedge \mathcal{PI}\}\ v.\mathrm{f} := \mathrm{E}\ \{\mathcal{PI}\}$

Precondition and effect of **unpack** E:

assert $\mathrm{E.inv} \wedge \neg \mathrm{E.com}$;

$\mathrm{E.inv}$:= false; **forall** o **with** $\mathrm{o.own} = \mathrm{E}$ **do** $\mathrm{o.com}$:= false;

Last auxiliary field for ownership discipline:

$\mathrm{com} : \textbf{boolean} := \mathit{false}$

$\mathcal{PI}: \ (\forall o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\ ) \wedge$

$(\forall o \bullet o.inv \Rightarrow (\forall p \bullet p.own = o \Rightarrow p.com\ )) \wedge$

$(\forall o \bullet o.com \Rightarrow o.inv\ )$

Absence of interf., as a precond.: $\{\neg v.inv \wedge \mathcal{PI}\}\ v.f := E\ \{\mathcal{PI}\}$

Precondition and effect of **unpack** E:

assert $E.inv \wedge \neg E.com$;

$E.inv$:= false; **forall** o **with** $o.own = E$ **do** $o.com$:= false;

Precondition and effect of **pack** E:

assert $\neg E.inv \wedge \mathcal{I}_{type(E)}(E)$;

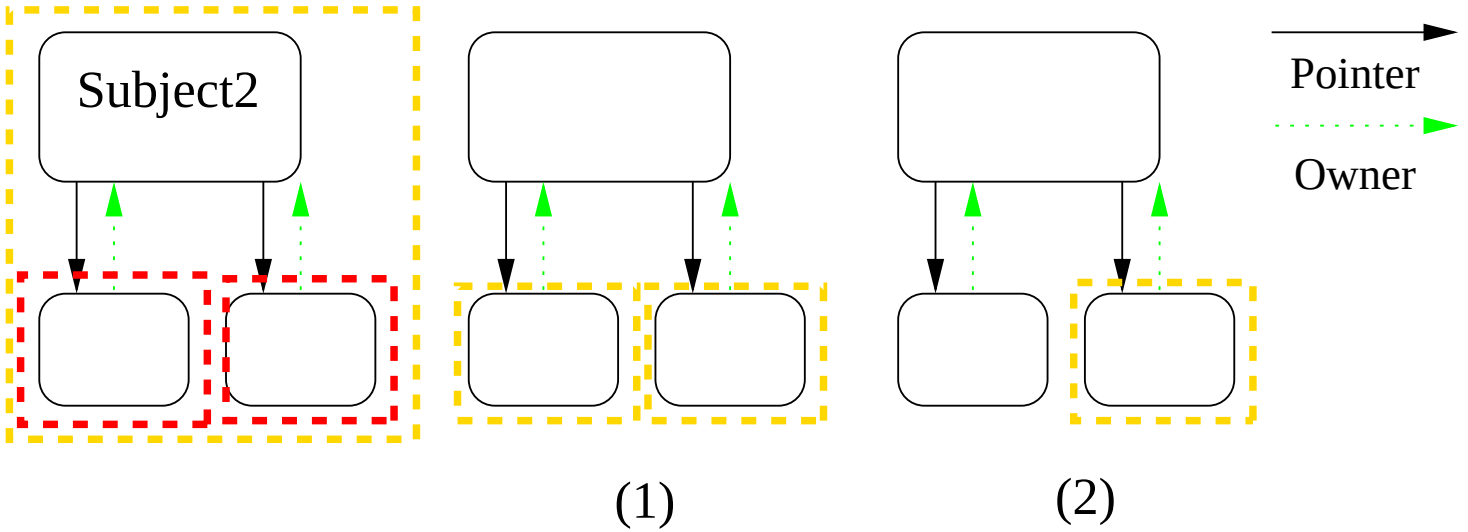
$E.inv$:= true; **forall** o **with** $o.own = E$ **do** $o.com$:= true;

class Subject2 . . . **method** m() { **assert** inv∧¬comm

unpack self; (1)

unpack x; (2)

x.value:=x.value+1; . . .

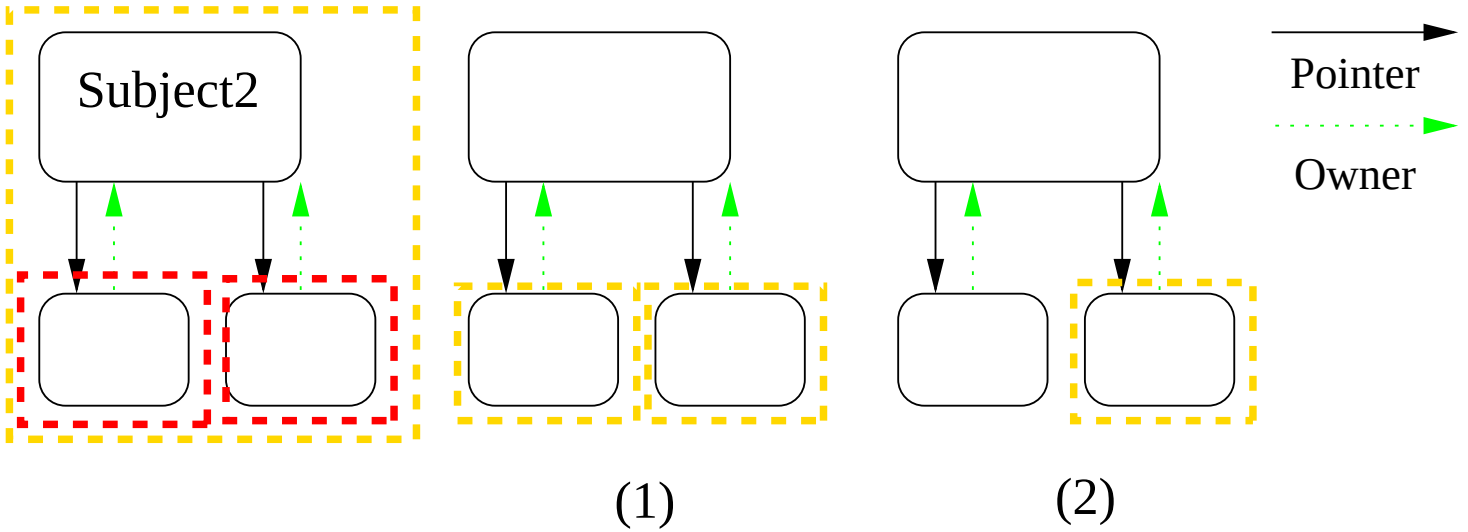


class Subject2 . . . **method** m() { **assert** inv∧¬comm

unpack self; (1)

unpack x; (2)

x.value:=x.value+1; . . .



Encapsulation is hierarchical. But hierarchy is mutable. . .

Ownership transfer

$\{\neg v.\mathrm{inv} \wedge \neg \mathrm{E}.\mathrm{inv} \wedge \mathcal{PI}\}\ v.\mathrm{own} := \mathrm{E}\ \{\mathcal{PI}\}$

Special command **setowner**; *only manipulates auxiliary state, like* **unpack/pack**.

Ownership transfer

$\{\neg v.\mathrm{inv} \wedge \neg \mathrm{E}.\mathrm{inv} \wedge \mathcal{PI}\}\ v.\mathrm{own} := \mathrm{E}\ \{\mathcal{PI}\}$

Special command **setowner**; *only manipulates auxiliary state, like* **unpack/pack**.

State-based encapsulation (vs. type systems):

- *avoids restriction on existence or reading of references*
- *allows transfer of objects across boundaries*
- *examples: lexer/stream, AST (into); tasks (between); database connections, malloc/free (in and out)*

Subclassing

$o.inv$ is a classname and $o.own$ is (object,classname).

$$\mathcal{PI}: \ (\forall o \bullet o.inv \geq type(o)\ ) \wedge$$
$$(\forall o \bullet o.inv \leq C \Rightarrow \mathcal{I}_C(o)\ ) \wedge$$
$$(\forall o \bullet o.inv \leq C \Rightarrow (\forall p \bullet p.own = (o, C) \Rightarrow p.com\ )) \wedge$$
$$(\forall o \bullet o.com \Rightarrow o.inv\ )$$

Precondition and effect of **pack** E **to** C:

assert $E.inv = super(C) \wedge \mathcal{I}_C(E)$;

$E.inv$:= C; **forall** o **with** $o.own = E$ **do** $o.com$:= true;

Stateful encapsulation I.

Def: S is *properly annotated* iff each **pack**, **unpack**, **setowner**, and field update has stipulated precondition.

Theorem: $\{\mathcal{PI}\}\ S\ \{\mathcal{PI}\}$ for any properly annotated S.

Corollary: $\dfrac{\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}}$ for all $S, \mathcal{P}, \mathcal{Q}$

Proof: Using a straightforward denotational semantics for a sequential language with mutually recursive class declarations and methods etc.

Stateful encapsulation II –sharing.

A List owns its nodes. A node does not own its neighbors.

class List {
  head: ListNode;
  invariant self.head=null $\vee$ self.head.prev=null; ...}
class ListNode {
  next, prev: ListNode;
  invariant self.next=null
    $\vee$ (self.next.prev=self $\wedge$ self.next.own=self.own); ...}

Stateful encapsulation II –sharing.

A List owns its nodes. A node does not own its neighbors.

class List {
  head: ListNode;
  invariant self.head=null $\vee$ self.head.prev=null; ...}
class ListNode {
  next, prev: ListNode;
  invariant self.next=null
    $\vee$ (self.next.prev=self $\wedge$ self.next.own=self.own); ...}

Decentralized invariants express acyclicity without induction.
Well behaved interaction —sharing, not separation.

Absence of interference, as a precondition:

$\{\neg v.\mathrm{in}v \wedge$

$\quad (\,\forall\, \mathrm{o} \bullet$ "$\mathcal{I}_{\mathrm{C}}(\mathrm{o})$ depends on $v.\mathrm{f}$" $\Rightarrow \neg \mathrm{o}.\mathrm{in}v \vee \mathcal{I}_{\mathrm{C}}(\mathrm{o})[\mathrm{E}/v.\mathrm{f}]\,)\}$

$v.\mathrm{f} := \mathrm{E}$

Absence of interference, as a precondition:

$\{\neg v.\mathrm{inv} \wedge$

$\quad (\,\forall\, o \bullet$ "$\mathcal{I}_C(o)$ depends on $v.f$" $\Rightarrow \neg o.\mathrm{inv} \vee \mathcal{I}_C(o)[E/v.f]\,)\}$

$v.f := E$

Auxiliary field for friendship discipline:

$\mathrm{deps} :$ **set of** $\mathrm{Object} := \varnothing$

$(\,\forall\, o \bullet o \in v.\mathrm{deps} \Rightarrow \neg o.\mathrm{inv} \vee \mathcal{I}_C(o)[E/v.f]\,)$

Absence of interference, as a precondition:

$\{\neg v.\mathrm{inv} \wedge$

$\quad (\,\forall\, \mathrm{o} \bullet$ "$\mathcal{I}_{\mathrm{C}}(\mathrm{o})$ depends on $v.\mathrm{f}$" $\Rightarrow \neg \mathrm{o}.\mathrm{inv} \vee \mathcal{I}_{\mathrm{C}}(\mathrm{o})[\mathrm{E}/v.\mathrm{f}]\,)\}$

$v.\mathrm{f} := \mathrm{E}$

Auxiliary field for friendship discipline:

$\mathrm{deps} :$ **set of** $\mathrm{Object} := \varnothing$

$(\,\forall\, \mathrm{o} \bullet \mathrm{o} \in v.\mathrm{deps} \Rightarrow \neg \mathrm{o}.\mathrm{inv} \vee \mathcal{I}_{\mathrm{C}}(\mathrm{o})[\mathrm{E}/v.\mathrm{f}]\,)$

Admissibility: when $\mathcal{I}_{\mathrm{C}}(\mathrm{o})$ *depends on* $\mathrm{p}.\mathrm{f}$ *then either* $\mathrm{o} = \mathrm{p}$, $\mathrm{o} \preceq \mathrm{p}$, *or* $\mathrm{p} = \mathrm{o}.\mathrm{g}$ *for some declared pivot* g *and* $\mathrm{o} \in \mathrm{p}.\mathrm{deps}$

Absence of interference, as a precondition:

$\{\neg v.\mathrm{inv} \wedge$

$\quad (\,\forall\, o \bullet$ "$\mathcal{I}_C(o)$ depends on $v.f$" $\Rightarrow \neg o.\mathrm{inv} \vee \mathcal{I}_C(o)[E/v.f]\,)\}$

$v.f := E$

Auxiliary field for friendship discipline:

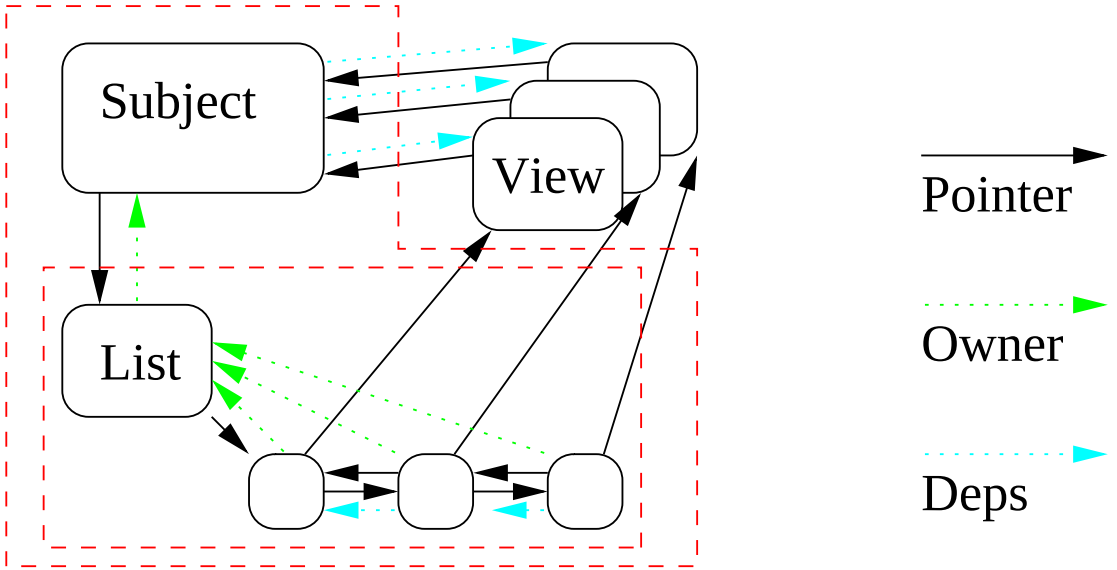
deps : **set of** $\mathrm{Object} := \varnothing$

$(\,\forall\, o \bullet o \in v.\mathrm{deps} \Rightarrow \neg o.\mathrm{inv} \vee \mathcal{I}_C(o)[E/v.f]\,)$

Admissibility: when $\mathcal{I}_C(o)$ *depends on* $p.f$ *then either* $o = p$, $o \preceq p$, *or* $p = o.g$ *for some declared pivot* g *and* $o \in p.\mathrm{deps}$

Abstract from $\mathcal{I}_C(o)[E/v.f]$ *as* $\mathcal{U}_C(o, v, E)$.

Obligation: $\{\mathcal{I}_C(\mathrm{self}) \wedge \mathcal{U}(\mathrm{self}, g, \mathrm{val})\}\ \mathrm{self}.g.f := \mathrm{val}\ \{\mathcal{I}_C(\mathrm{self})\}$



class Subject3 { val: int; (*state*) vsn: int; (*for sync*)

friend View **reads** vsn, val; ... }

class View { version: int; **private** s: Subject; cache: int;

invariant s.vsn$-1 \leq$version$\leq$s.vsn $\wedge$

(s.vsn=version $\Rightarrow$ cache=s.val);

guard s.vsn:=α **by** $\alpha - 1 \leq$ version $\leq \alpha$; ... }

Encapsulation: rep. independence

class Subject2 {

  private x: Integer := **new** Integer(0);

  private y: Integer := **new** Integer(1);

  invariant $\mathcal{I}$(self) where $\mathcal{I}$(o) = o.x.val$<$o.y.val

  method m() { self.x.incr(); self.y.incr(); }

class Subject2 { // Alternate version

  private x: **int** := 0;

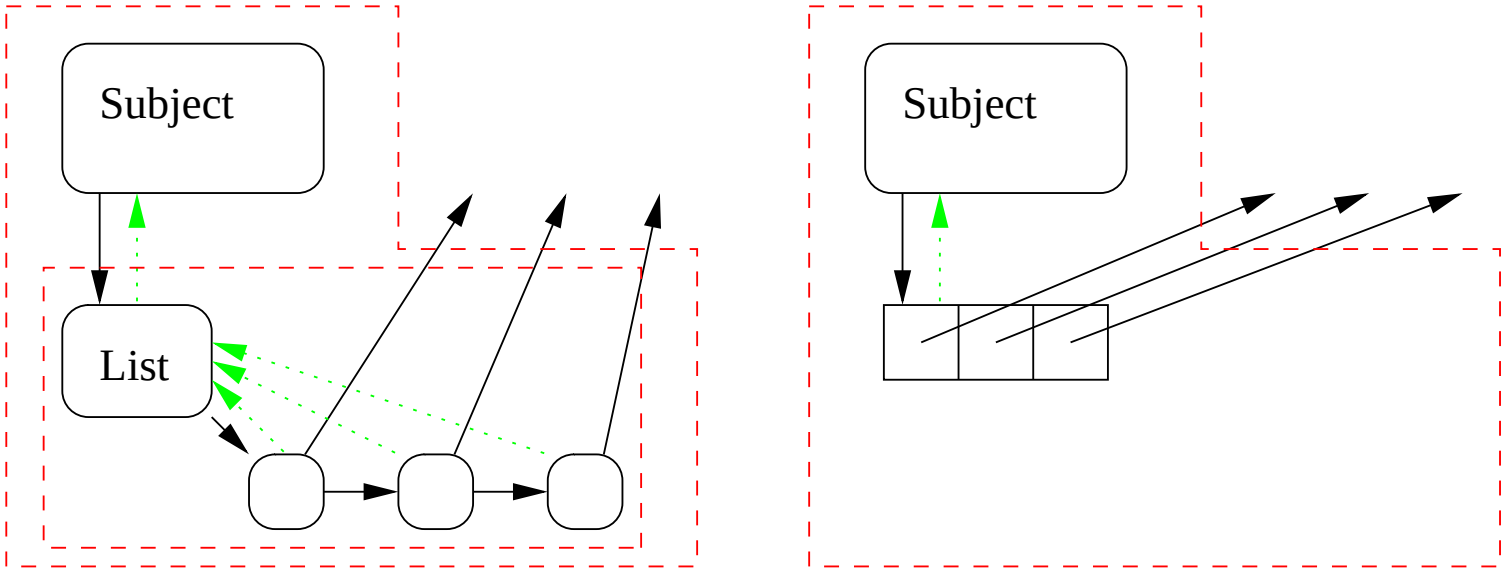
  private z: **int** := 1;

  invariant $\mathcal{I}$(self) where $\mathcal{I}$(o) = 0$<$o.z

  method m() { self.x := self.x + 1; }

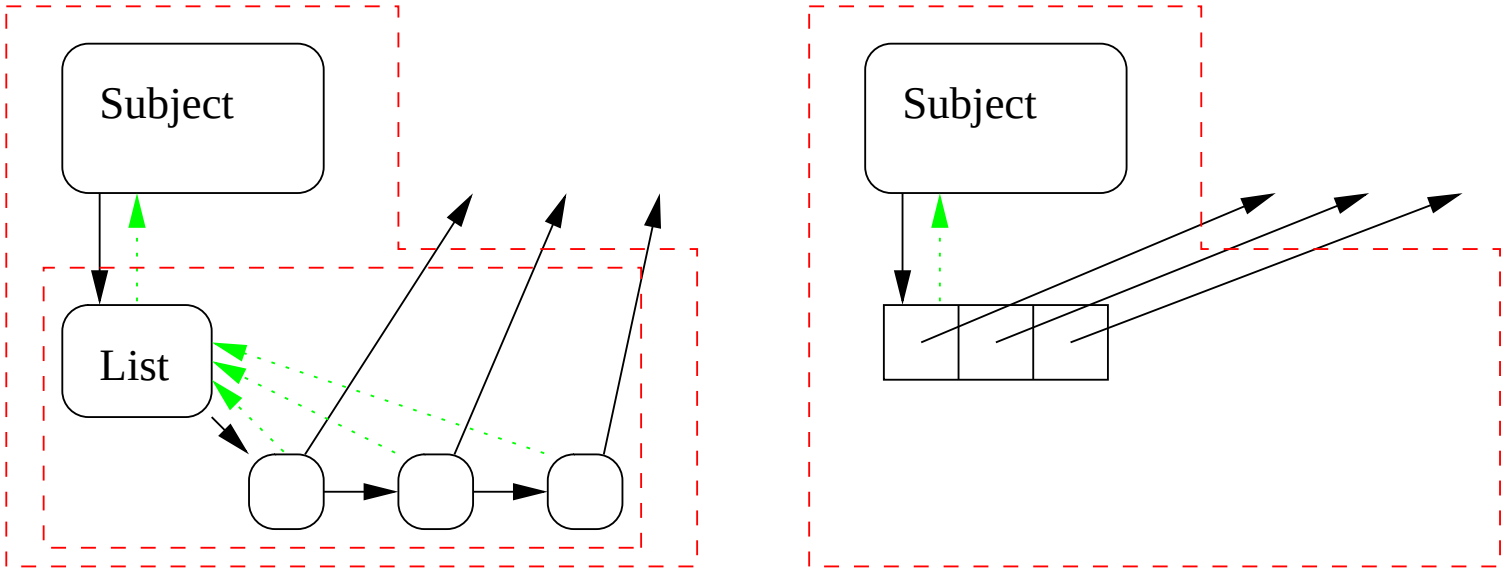
Coupling relation: o.x.val = o'.x $\wedge$ o.y.val$-$o.x.val = o'.z

Towards modular coupling



Coupling relation: elts(o.list) = elts(o'.arr)

Towards modular coupling

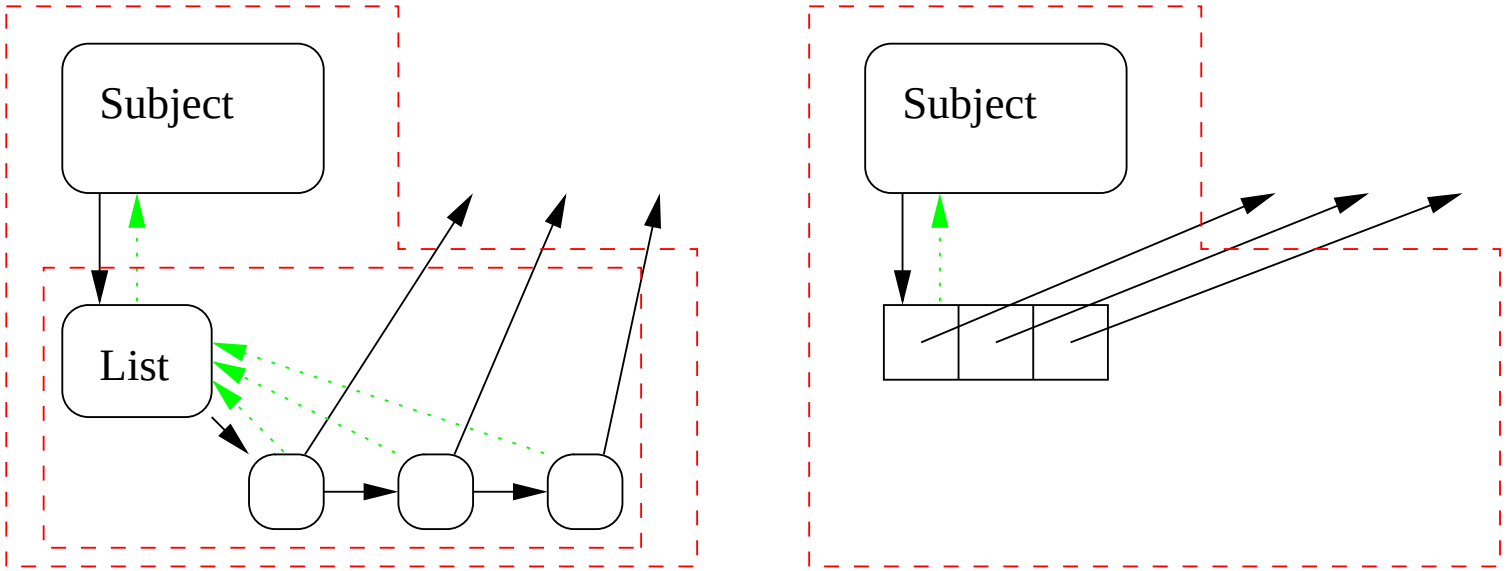


Coupling relation: elts(o.list) = elts(o'.arr)

Partition $\mathrm{h} = \mathrm{Xh} * \mathrm{Island}_1 * \ldots * \mathrm{Island}_k$

where $\mathrm{Island}_i =$ (instance o_i of Subject) $*$ (owned by o_i).

Towards modular coupling



Coupling relation: elts(o.list) = elts(o'.arr)

Partition $h = Xh * Island_1 * \ldots * Island_k$

where $Island_i$ = (instance o_i of Subject) $*$ (owned by o_i).

Then $h \models \mathcal{I}(o_i)$ iff $Island_i \models \mathcal{I}(o)$ for admissible $\mathcal{I}$.

Coupling

Heaps h, h' coupled just if there are same-length partitions

$\mathrm{h} = \mathrm{Xh} * \mathrm{Island}_1 * \ldots \mathrm{Island}_k$

$\mathrm{h}' = \mathrm{Xh}' * \mathrm{Island}'_1 * \ldots \mathrm{Island}'_k$

Coupling

Heaps h, h' coupled just if there are same-length partitions

$h = Xh * Island_1 * \ldots Island_k$

$h' = Xh' * Island'_1 * \ldots Island'_k$

and for each i we have

$o_i.inv = o'_i.inv$ and

$o_i.inv$ implies $Island_i$ coupled with $Island'_i$

(by the given single-instance coupling).

Coupling

Heaps h, h' coupled just if there are same-length partitions

$h = Xh * Island_1 * \ldots Island_k$

$h' = Xh' * Island'_1 * \ldots Island'_k$

and for each i we have

$o_i.inv = o'_i.inv$ and

$o_i.inv$ implies $Island_i$ coupled with $Island'_i$

(by the given single-instance coupling).

Moreover $Xh \sim Xh'$ (modulo bijective renaming).

Coupling

Heaps h, h' coupled just if there are same-length partitions

$h = Xh * Island_1 * \ldots Island_k$

$h' = Xh' * Island'_1 * \ldots Island'_k$

and for each i we have

$o_i.inv = o'_i.inv$ and

$o_i.inv$ implies $Island_i$ coupled with $Island'_i$

(by the given single-instance coupling).

Moreover $Xh \sim Xh'$ (modulo bijective renaming).

Identity on visible state (fields in Xh, interface of o_i).

Abstraction theorem

Theorem If the induced coupling is a *simulation*, i.e., is preserved by the methods of the revised class A, then it is preserved by all properly annotated contexts.

Abstraction theorem

Theorem If the induced coupling is a *simulation*, i.e., is preserved by the methods of the revised class A, then it is preserved by all properly annotated contexts.

Corollary: proof of program equivalence.

Abstraction theorem

Theorem If the induced coupling is a *simulation*, i.e., is preserved by the methods of the revised class A, then it is preserved by all properly annotated contexts.

Corollary: proof of program equivalence.

Reentrant callbacks and invariants/simulations: a method that does not require inv cannot rely on $\mathcal{I}$ but may later reestablish $\mathcal{I}$ —need modifies-specs of called methods.

Adaptations for simulation

–read fields of objects owned by an A: only in code of A

–Pack self to A: only in code of A, as it restores coupling.

–owner transfer: export of A's reps only by code of A

Stronger alias control than for invariants, but only for A.

Observational purity

```
class View {
   version: int; private s: Subject; cache: int; pending: bool
   notify() { pending:=true; }
   pure get(): int {
      if pending then cache:=s.val endif; return cache; ... }
```

Observational purity

```
class View {
   version: int; private s: Subject; cache: int; pending: bool
   notify() { pending:=true; }
   pure get(): int {
      if pending then cache:=s.val endif; return cache; ... }
```

Use simulation to prove equivalent to an implementation in which get is strongly pure.

Observational purity

```
class View {
   version: int; private s: Subject; cache: int; pending: bool
   notify() { pending:=true; }
   pure get(): int {
      if pending then cache:=s.val endif; return cache; ... }
```

Use simulation to prove equivalent to an implementation in which get is strongly pure.

skip $\approx$ **assert** x.get() = 0 outside class View

Conclusion

- Discipline for control of dependence for object invariants. Controls use of pointers rather than their existence.
- Handles difficult design patterns that are common in practice, including cooperative sharing.
- No committment to particular program logic or verification system.
- Uses verification conditions; not special type annotation but not fully automated.

Future work

- *precise* comparison with Separation Logic:

$$\frac{\{\mathcal{P}\}\ \mathtt{m}\ \{\mathcal{Q}\} \vdash \{\mathcal{P}'\}\ S\ \{\mathcal{Q}'\}}{\{\mathcal{P} * \mathcal{I}\}\ \mathtt{body}\ \{\mathcal{Q} * \mathcal{I}\} \vdash \{\mathcal{P}' * \mathcal{I}\}\ S\ \{\mathcal{Q}' * \mathcal{I}\}}$$

- implementation, case studies, concurrency —Spec# project
- friends generalized to multi-class patterns
- integrate with ownership typing, extend to concurrency and refinement
- machine check soundness proofs

Related work

- Leino et al [JoT, ECOOP04, CASSIS04, Statics]
- O'Hearn et al [POPL04]; Mijajlović et al [FSTTCS 04] –static modularity for separation logic
- Hongseok Yang [TCS?] –relational sep. logic
- full logic and mechanization —Pierik and de Boer

Advert

Seeking PhD student or postdoc to develop these ideas in context of JML, a specification language used by ESC/Java and several other systems e.g. smartcard verif. (Joint project with Iowa State (Gary Leavens) and UFPE, Recife, Brazil.)

References

Barnett, DeLine, Fähndrich, Leino, Schulte: *Verifi cation of object-oriented programs with invariants* (JoT ’04)

Leino, Müller: *Object invariants in dynamic contexts* (ECOOP’04)

Barnett, DN: *Friends need a bit more* (MPC’04); (LICS’04)

Pierik, Clarke, de Boer: *Creational invariants* (FTfJP’04)

O’Hearn, Yang, Reynolds: *Separation and Info Hiding* (POPL’04)

Banerjee and DN: *State based ownership and encapsulation for generic classes*

DN: *Observational purity and encapsulation*

Calcagno, O’Hearn, Bornat: Program logic and equiv. vs. garbage

(TCS03)