Towards imperative modules: reasoning about invariants and sharing of mutable state

David A. Naumann

Joint work with Mike Barnett and Anindya Banerjee

Stevens Institute of Technology

Supported by NSF CCR-0208984, CCF-0429894, and Microsoft.

Outline

- Difficulty in reasoning about object invariants due to callbacks and heap sharing —programmer's view
- —logician's view
- The boogie and friends disciplines: state based encapsulation (with Mike Barnett [LICS])
- Representation independence (with Anindya Banerjee)
- 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 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 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 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)

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

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)

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

$$\vdots$$

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

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)

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

$$\vdots$$

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

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.

Auxiliary field to make explicit when invariant holds:

inv : **boolean** := false

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

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** $:= \mathit{false}$

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

Last auxiliary field for ownership discipline:

com : **boolean** $:= \mathtt{false}$

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

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:

com : **boolean** := 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.: $\{\neg \mathrm{v.inv} \wedge \mathcal{PI}\}\ \mathrm{v.f} := \mathrm{E}\ \{\mathcal{PI}\}$

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:

com : **boolean** := false

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

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

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

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

Precondition and effect of **unpack** E:

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

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

Precondition and effect of **pack** E:

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

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

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** *to highlight that it 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 (in and out)*

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

Justifies rule: $$\frac{\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}}$$

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

Stateful encapsulation II –friends.

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 –friends.

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 but not ownership.

Absence of interference, as a precondition:

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

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

$v.f := E$

Absence of interference, as a precondition:

$\{\neg v.inv \wedge$

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

$v.f := E$

Auxiliary field for friendship discipline:

$deps$: **set of** $Object := \varnothing$

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

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* $o \in p.\mathrm{deps}$

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

Program equivalence: two-state invars

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: Integer := **new** Integer(1);
  invariant $\mathcal{I}$(self) where $\mathcal{I}$(o) = 0$<$o.z.val
  method m() { self.x := self.x + 1; }

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

Towards simulation: admissibility revisited

Let o be an instance of the class A to be revised.

Partition $\mathrm{h} = A\mathrm{h} * \mathrm{R}\mathrm{h} * \mathrm{X}\mathrm{h}$, where

$\mathrm{dom}(A\mathrm{h}) = \{\mathrm{o}\}$ and

$\mathrm{dom}(\mathrm{R}\mathrm{h}) =$ the set of objects transitively owned by o in h.

Then $\mathrm{h} \models \mathcal{I}_A(\mathrm{o})$ iff $A\mathrm{h} * \mathrm{R}\mathrm{h} \models \mathcal{I}_A(\mathrm{o})$.

Towards simulation: admissibility revisited

Let o be an instance of the class A to be revised.

Partition $h = Ah * Rh * Xh$, where
$dom(Ah) = \{o\}$ and
$dom(Rh) =$ the set of objects transitively owned by o in h.
Then $h \models \mathcal{I}_A(o)$ iff $Ah * Rh \models \mathcal{I}_A(o)$.

$\mathcal{PI}$ implies: If we choose top-level instances $o_1 \ldots o_k$ of A in h, have for $o_i.inv \Rightarrow Ah_i * Rh_i \models \mathcal{I}_A(o_i)$ for all i
where $h = Xh * (Ah_1 * Rh_1) * \ldots (Ah_k * Rh_k)$.

Coupling for two versions of $\mathcal{A}$

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

$\mathrm{h} = \mathrm{Xh} * (\mathcal{A}\mathrm{h}_1 * \mathrm{Rh}_1) * \ldots (\mathcal{A}\mathrm{h}_k * \mathrm{Rh}_k)$

$\mathrm{h}' = \mathrm{Xh}' * (\mathcal{A}\mathrm{h}'_1 * \mathrm{Rh}'_1) * \ldots (\mathcal{A}\mathrm{h}'_k * \mathrm{Rh}'_k)$

Coupling for two versions of A

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

$h = Xh * (Ah_1 * Rh_1) * \ldots (Ah_k * Rh_k)$

$h' = Xh' * (Ah'_1 * Rh'_1) * \ldots (Ah'_k * Rh'_k)$

such that for each pair, $o_i.inv = o'_i.inv$ and $o_i.inv$ implies $(Ah_i * Rh_i)$ relates to $(Ah'_i * Rh'_i)$ by a given coupling.

Coupling for two versions of A

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

$h = Xh * (Ah_1 * Rh_1) * \ldots (Ah_k * Rh_k)$

$h' = Xh' * (Ah'_1 * Rh'_1) * \ldots (Ah'_k * Rh'_k)$

such that for each pair, $o_i.inv = o'_i.inv$ and $o_i.inv$ implies $(Ah_i * Rh_i)$ relates to $(Ah'_i * Rh'_i)$ by a given coupling.

Moreover Xh corresponds to Xh' (modulo bijective renaming of locations).

Coupling for two versions of A

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

$\mathrm{h} = \mathrm{Xh} * (\mathrm{Ah}_1 * \mathrm{Rh}_1) * \ldots (\mathrm{Ah}_k * \mathrm{Rh}_k)$

$\mathrm{h}' = \mathrm{Xh}' * (\mathrm{Ah}'_1 * \mathrm{Rh}'_1) * \ldots (\mathrm{Ah}'_k * \mathrm{Rh}'_k)$

such that for each pair, $\mathrm{o}_i.inv = \mathrm{o}'_i.inv$ and $\mathrm{o}_i.inv$ implies $(\mathrm{Ah}_i * \mathrm{Rh}_i)$ relates to $(\mathrm{Ah}'_i * \mathrm{Rh}'_i)$ by a given coupling.

Moreover Xh corresponds to Xh' (modulo bijective renaming of locations).

Identity on visible state (fields in Xh, interface of Ah_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 contexts.

If coupling holds at boundaries of A then outside it holds everywhere that inv does.

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 contexts.

If coupling holds at boundaries of A then outside it holds everywhere that inv does.

Reentrant callbacks and invariants: a method that does not require inv cannot rely on $\mathcal{I}$; that's all.

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 contexts.

If coupling holds at boundaries of A then outside it holds everywhere that inv does.

Reentrant callbacks and invariants: a method that does not require inv cannot rely on $\mathcal{I}$; that's all.

Reentrant callbacks and simulation: a method that does not require inv must still preserve —how? (vs. invariant case where some precondition can help)? Need modifies spec.

(e.g., callbacks from notify can inspect the Subject but not alter the datastruct tracking Observers).

(e.g., callbacks from notify can inspect the Subject but not alter the datastruct tracking Observers).

(e.g., callbacks from notify can inspect the Subject but not alter the datastruct tracking Observers).

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. No restrictions on heap structure.
- No committment to particular program logic or verification system.
- Uses verification conditions; not special type annotation but not fully automated.

Related work

- Leino et al [JoT, ECOOP04, CASSIS04] –Boogie, Spec# with concurrency
- O'Hearn et al [POPL04]; Mijajlović et al [FSTTCS 04] –static modularity for separation logic
- Parkinson & Bierman [POPL05] –instantiable abstraction in sep. logic using scope of predicate definitions
- Hongseok Yang [TCS?] –relational sep. logic
- full logic and mechanization —Pierik and de Boer

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 and case studies —Spec# project
- friends and subclassing; generalization to multi-class patterns —Barnett and Naumann
- integrate with ownership typing, extend simu to concurrent —Banerjee and Naumann
- machine check soundness proof —Naumann

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, Wolfram Schulte: *Verification of object-oriented programs with invariants* (Journal of Object Technology '04)
- Leino and Müller: *Object invariants in dynamic contexts* (ECOOP'04)
- Barnett and D.N.: *Friends need a bit more* (MPC'04)
- O'Hearn, Yang, Reynolds: *Separation and Info Hiding* (POPL'04)
- Banerjee and D.N.: *State based ownership and encapsulation for generic classes*