Use auxiliary state to express modular structure

Grand Challenge Workshop in Software Verification (ETAPS'05)

David A. Naumann

Stevens Institute of Technology

Hoboken, New Jersey, USA

Supported by NSF CCR-0208984 and CCF-0429894.

Outline

- Java-like programs are good challenge
- ✔ use auxiliary state to represent modular structure and reasoning discipline
- coupled tools: static analysis checks side conditions on rules for modular reasoning
- possible challenge codes and specifications

Java: good point in lang. des. space

- ✘ not theoretically elegant; words: “method”, “this”
- ✔ express higher order design patterns (e.g., map/visitor)
- ✔ “defunctionalized”—simple adequate semantic models
- ✔ other semantic complexity absent: aliased vars/params, reference to locals in enclosing scope; method update
- ✘ use of (module scoped) globals, shared objects (heap), even reflection
- ✔ nominal typing: convenient hook for specs and rules

Modular reasoning—naive view

Package collects interrelated classes (unit of scope).

Class instances provide some abstraction, maintaining state invariants (visible and internal, e.g., Subject has set of registered Views, stored in an array) .

Method specification describes operation in terms of the abstraction.

Method implementation uses other abstractions and is verified with respect to their specifications, using the internal invariant.

Challenge: non-heirarchical control

Method specification describes operation in terms of abstraction that involves upcalls to clients (e.g., sensor with *set* of views; when sensor $\geq \mathrm{o}$.*thresh*, remove o from view set and notify o)

Heirarchical layering is violated: reentrant callback —e.g., notified view queries the sensor value (ok) or enumerates the view set (ouch!).

Challenge: non-heirarchical control

Method specification describes operation in terms of abstraction that involves upcalls to clients (e.g., sensor with *set* of views; when sensor $\geq$ o.*thresh*, remove o from view set and notify o)

Heirarchical layering is violated: reentrant callback —e.g., notified view queries the sensor value (ok) or enumerates the view set (ouch!).

Broken internal invariant: array contains those views for which threshold not met.
Subject and View both abstract w.r.t. the other.

Non-heirarchical control—solutions?

- ✘ Establish invariant before every method call.
- ✘ Temporal specification of allowed calling pattern.
- ✘ Use locks to prevent reentrance.
- ✔ Precondition—e.g, for the enumerate method: notify not in progress, or, invariant is in force.

$\{\mathcal{I}(snsr)\}\ snsr.enum()$, $\quad \{\neg\mathcal{I}(snsr)\}\ view.notify(snsr)$

Non-heirarchical control—solutions?

- ✘ Establish invariant before every method call.
- ✘ Temporal specification of allowed calling pattern.
- ✘ Use locks to prevent reentrance.
- ✔ Precondition—e.g, for the enumerate method: notify not in progress, or, invariant is in force.

$\{\mathcal{I}(snsr)\}\ snsr.enum()$, $\{\neg\mathcal{I}(snsr)\}\ view.notify(snsr)$

Hiding info about $\mathcal{I}$: typestate [DeLine,Fähndrich], opaque predicate [Parkinson&Bierman,Birkedal&Torp-Smith], (observationally) pure methods, auxiliary field [Leino et al,Müller].

Ghost (auxiliary) field: $\mathrm{inv} : \{\mathrm{mut}, \mathrm{valid}, \mathrm{cmtd}\} := \mathrm{mut}$

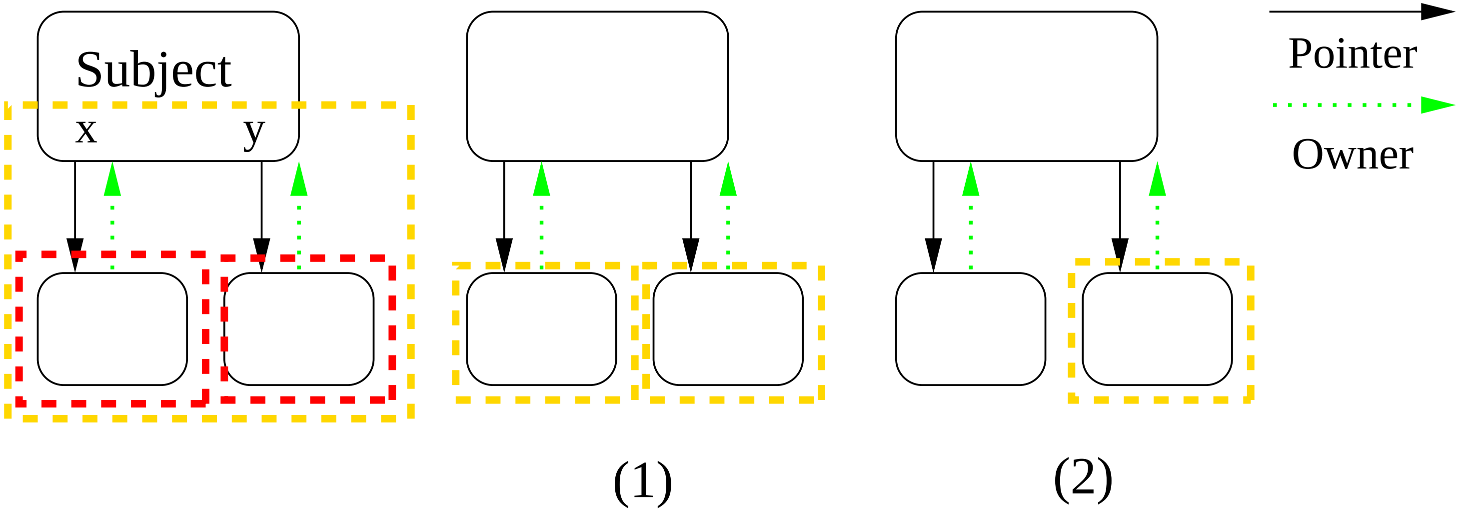
Program invariant: $(\, \forall\, o \mid o.\mathrm{inv} = \mathrm{mut} \vee \mathcal{I}_(o)\ )$

Ghost (auxiliary) field: $\mathrm{inv} : \{\mathrm{mut}, \mathrm{valid}, \mathrm{cmtd}\} := \mathrm{mut}$

Program invariant: $(\,\forall\, o \mid o.\mathrm{inv} = \mathrm{mut} \vee \mathcal{I}_(o)\ )$

Protocol to control crossing of encapsulation boundaries:

assert(self.inv= valid); **unpack** self; (1) **unpack** x; (2)



Challenge: sharing and encapsulation

Encapsulation by lexical scope: if field f is local to class K and the invariant, $\mathcal{I}_{\mathrm{K}}$, of K depends only on f then the invariant is maintained by all commands outside class K.

Challenge: sharing and encapsulation

Encapsulation by lexical scope: if field f is local to class K and the invariant, $\mathcal{I}_{\mathrm{K}}$, of K depends only on f then the invariant is maintained by all commands outside class K.

Frame Rule: $$\frac{\{\mathcal{P}\}\,\mathrm{c}\,\{\mathcal{Q}\} \qquad \mathrm{c} \text{ does not interfere with } \mathcal{I}}{\{\mathcal{P} \wedge \mathcal{I}\}\,\mathrm{c}\,\{\mathcal{Q} \wedge \mathcal{I}\}}$$

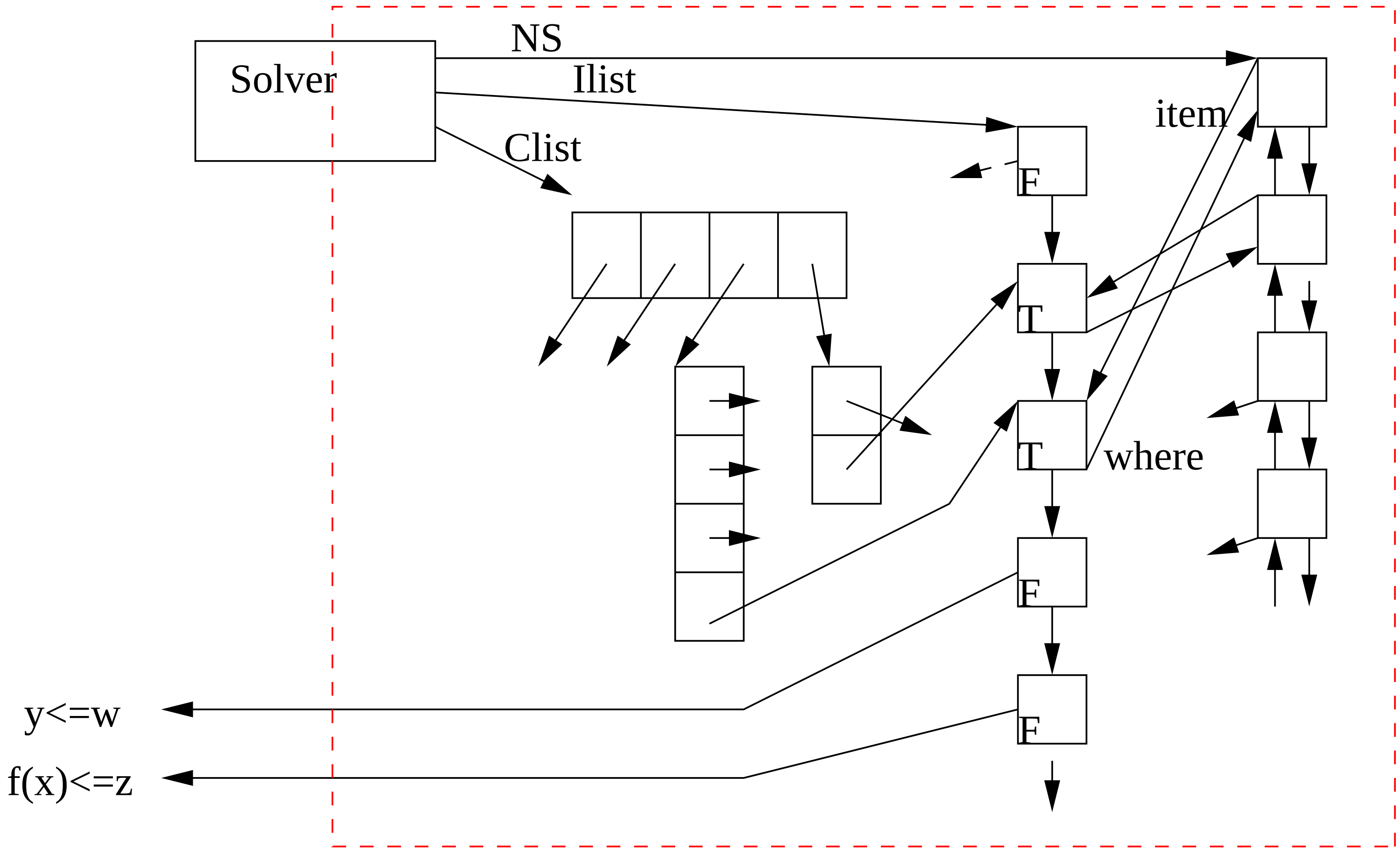
Challenge: sharing and encapsulation

Encapsulation by lexical scope: if field f is local to class K and the invariant, $\mathcal{I}_{\mathsf{K}}$, of K depends only on f then the invariant is maintained by all commands outside class K.

Frame Rule: $$\frac{\{\mathcal{P}\}\ \mathrm{c}\ \{\mathcal{Q}\} \qquad \mathrm{c} \text{ does not interfere with } \mathcal{I}}{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathrm{c}\ \{\mathcal{Q} \wedge \mathcal{I}\}}$$

If $\mathcal{I}$ depends on heap objects, use separation:

$$\frac{\{\mathcal{P}\}\ \mathrm{m}\ \{\mathcal{Q}\} \vdash \{\mathcal{P}'\}\ \mathrm{c}\ \{\mathcal{Q}'\}}{\{\mathcal{P} * \mathcal{I}\}\ \mathrm{body}_{\mathrm{m}}\ \{\mathcal{Q} * \mathcal{I}\} \vdash \{\mathcal{P}' * \mathcal{I}\}\ \mathrm{c}\ \{\mathcal{Q}' * \mathcal{I}\}}$$



$$\mathcal{I}_{\mathsf{Subject}}: \quad \forall p \in \mathsf{NS.next}* \cdot (\mathsf{p.next} = \mathsf{null} \vee \mathsf{p.next.prev} = \mathsf{p})$$

$$\wedge(\mathsf{p.item} \in \mathsf{Ilist.next}*) \wedge (\exists x, j \cdot \mathsf{Clist}[x][j] = \mathsf{p.item}) \ldots$$

Object ownership

Ownership types [Clarke,Aldrich,Boyapati,. . .]: static ownership forest, constrain existence of references.

Ownership in ghost field [Leino,. . .]: stateful discipline to control use of references (previous slide; cf. locks).

Object ownership

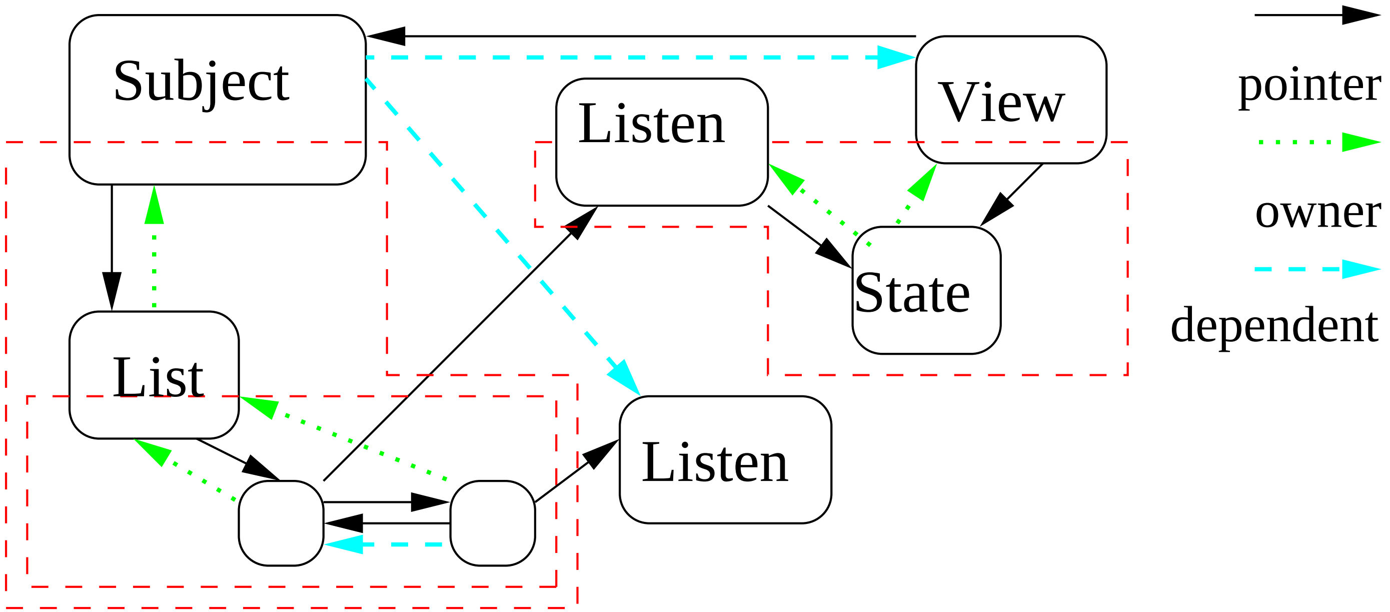
Ownership types [Clarke,Aldrich,Boyapati,. . .]: static ownership forest, constrain existence of references.

Ownership in ghost field [Leino,. . .]: stateful discipline to control use of references (previous slide; cf. locks).

Challenges:

- Shared ownership, e.g., iterators.
- Transfer, e.g., task queues (between); lexer/stream, AST (into); database connections, malloc/free (in and out).
- Objects decompose: inherited fields, “future” subclasses.

Challenge: cooperative sharing



$$\mathcal{I}_{\text{Node}}: \quad next = null \lor next.prev = self$$

$$\mathcal{I}_{\text{View}}: \quad subj.vsn = version \Rightarrow cache = s.val$$

Friendship discipline

Ghost field: $\mathsf{deps} : \textbf{set of } \mathsf{Object} := \varnothing$

Obligation (absence of interference, as a precondition):

$$\{\neg p.inv \land (\forall o \mid o \in p.deps \Rightarrow \neg o.inv \lor \underline{\mathcal{I}_C(o)[E/p.f]})\}$$
$$p.f := E$$

Friendship discipline

Ghost field: $\mathrm{deps} : \textbf{set of } \mathrm{Object} := \varnothing$

Obligation (absence of interference, as a precondition):

$$\{\neg p.inv \wedge (\forall o \mid o \in p.deps \Rightarrow \neg o.inv \vee \underline{\mathcal{I}_C(o)[E/p.f]})\}$$
$$p.f := E$$

Admissibility: when $\mathcal{I}_C(o)$ *depends on* $p.f$ *then either* $o = p$, o *transitively owns* p, *or* $o \in p.deps$

Friendship discipline

Ghost field: $\mathrm{deps} : \textbf{set of } \mathrm{Object} := \varnothing$

Obligation (absence of interference, as a precondition):

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

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

Admissibility: when $\mathcal{I}_C(\mathrm{o})$ *depends on* $\mathrm{p.f}$ *then either* $\mathrm{o} = \mathrm{p}$*,* o *transitively owns* p*, or* $\mathrm{o} \in \mathrm{p.deps}$

Abstract $\mathcal{I}_C(\mathrm{o})[\mathrm{E}/\mathrm{p.f}]$ *as declared "update guard"* $\mathcal{U}(\mathrm{o}, \mathrm{p}, \mathrm{E})$ *.*

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

Design/Verification Patterns

Specialized discipline for global invar. from local obligations.

Encap. from first-order assertions with auxiliary fields.

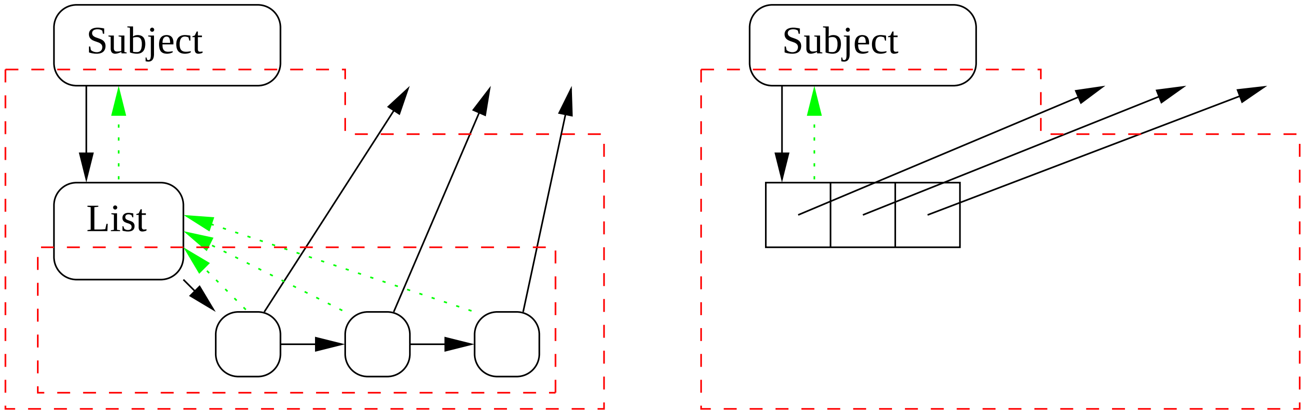
- interactive proof: specify and verify particular discipline
- automatic: check its application

Design/Verification Patterns

Specialized discipline for global invar. from local obligations.
Encap. from first-order assertions with auxiliary fields.

- interactive proof: specify and verify particular discipline
- automatic: check its application

Encapsulation for two-state invariants too [B&N ECOOP'05]:



Theory challenges

- encapsulation disciplines: cooperative sharing and separation— how much farther can we go using auxiliary state and standard logic?
- foundational logic for JML, encompassing concurrency, encapsulation disciplines (auxiliary state and scopes), behavioral subtyping
- why are heap regions second class? sep. log. needs quantification over predicates; why not *expressions* describing regions? [Aldrich domains/permissions; precise predicates for h.o. frame rule]
- specify the Observer and Visitor patterns (first order)

Challenge code: ad hoc net application

- small code and persistent state
- little legacy, development using disciplined language
- correctess is critical
- open, extensible systems: focus on resources/access [Verificard, MRG, Chander et al ESOP]

Correctness of closed system: just beyond current technology?

Open and rapidly evolving system: 5–10 year time frame?

Challenge code: ad hoc net application

- small code and persistent state
- little legacy, development using disciplined language
- correctess is critical
- open, extensible systems: focus on resources/access [Verificard, MRG, Chander et al ESOP]

Correctness of closed system: just beyond current technology?

Open and rapidly evolving system: 5–10 year time frame?

Susanne Wetzel—Bluetooth, sensors for physical therapy

Selected related work

- Clarke; Aldrich; Boyapati: owner. types [OOPSLA02,ECOOP04,. . .]
- Leino, Müller, et al: ghost owner, etc. [JoT, ECOOP04, CASSIS04,. . .]
- Pierik and de Boer: full logic and mechanization
- O'Hearn,Yang,Reynolds [POPL04]; Mijajlović et al [FSTTCS04]: static modularity for separation logic;
- Parkinson&Bierman [POPL05]: opaque predicate definition; Birkedal,Torp-Smith [ESOP05]: higher order separation logic
- DeLine, Fähndrich [Fugue]: opaque pre/post, static analysis
- Barnett et al; Weirich: info flow analysis for static encapsulation [FASE05,ICFP?]

```
class Master {
  time: int; invariant 0 ≤ time;    friend Clock reads time;
  tick(n: int){ assert inv = valid ∧ 0 ≤ n ;
                unpack self;   time += n;   pack self;
                assert time ≥ Old(time); }
  connect(c: Clock){ assert inv= valid;
                     unpack self; attach c;  pack self;
                     assert c ∈ self.deps; } }
class Clock {
  t: int;   m: Master;   invariant m ≠ null ∧ 0 ≤ t ≤ m.time;
                         guard m.time := α by m.time ≤ α;
  Clock(mast: Master) { assert mast ≠ null ∧ mast.inv= valid;
                        m:=mast; m.connect(self); t:=m.time; } }
```