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 \( \geq o.\text{thresh} \), 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 \( \geq o.\text{thresh} \), 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.

\[ \{ I(snsr) \} snsr.enum(), \quad \{ \neg I(snsr) \} view.notify(snsr) \]

Hiding info about \( I \): typestate [DeLine,Fähndrich], opaque predicate [Parkinson&Bierman,Birkedal&Torp-Smith], (observationally) pure methods, auxiliary field [Leino et al,Müller].
**Ghost (auxiliary) field:** \( \text{inv} : \{\text{mut, valid, cmtd}\} := \text{mut} \)

**Program invariant:** \( (\forall o \mid o.\text{inv} = \text{mut} \lor I(o) ) \)
**Ghost (auxiliary) field:** \( \text{inv} : \{\text{mut, valid, cmtd}\} := \text{mut} \)

**Program invariant:** \((\forall o \mid o.\text{inv} = \text{mut} \lor \mathcal{I}(o))\)

Protocol to control crossing of encapsulation boundaries:

\texttt{assert}(\ \text{self.inv= valid} \); \texttt{unpack} \ \text{self}; (1) \texttt{unpack} \ \text{x}; (2)
Challenge: sharing and encapsulation

Encapsulation by lexical scope: if field $f$ is local to class $K$ and the invariant, $I_K$, 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, $I_K$, of $K$ depends only on $f$ then the invariant is maintained by all commands outside class $K$.

Frame Rule: $\{P\} \ c \ \{Q\}$  \(c\) does not interfere with $I$

$$\{P \land I\} \ c \ \{Q \land I\}$$
Challenge: sharing and encapsulation

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

Frame Rule:

$$
\left\{ \mathcal{P} \right\} \text{ c } \left\{ \mathcal{Q} \right\} \quad \text{c does not interfere with } \mathcal{I} \\
\quad \Rightarrow \quad \left\{ \mathcal{P} \land \mathcal{I} \right\} \text{ c } \left\{ \mathcal{Q} \land \mathcal{I} \right\}
$$

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

$$
\left\{ \mathcal{P} \right\} \text{ m } \left\{ \mathcal{Q} \right\} \vdash \left\{ \mathcal{P}' \right\} \text{ c } \left\{ \mathcal{Q}' \right\} \\
\quad \Rightarrow \quad \left\{ \mathcal{P} \land \mathcal{I} \right\} \text{ body}_m \left\{ \mathcal{Q} \land \mathcal{I} \right\} \vdash \left\{ \mathcal{P}' \land \mathcal{I} \right\} \text{ c } \left\{ \mathcal{Q}' \land \mathcal{I} \right\}
$$
$\mathcal{I}_{Subject} : \forall p \in NS.\mathit{next}^* \cdot (p.\mathit{next} = \mathit{null} \lor p.\mathit{next}.\mathit{prev} = p) \\
\land (p.\mathit{item} \in \mathit{Ilist}.\mathit{next}^*) \land (\exists x, j \cdot \mathit{Clist}[x][j] = p.\mathit{item}) \ldots$
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

\[ I_{\text{Node}} : \quad \text{next} = \text{null} \lor \text{next.prev} = \text{self} \]

\[ I_{\text{View}} : \quad \text{subj.vsn} = \text{version} \Rightarrow \text{cache} = \text{s.val} \]
Friendship discipline

Ghost field:  \( \text{deps : set of Object := } \emptyset \)

Obligation (absence of interference, as a precondition):
\[
\{ \neg p.\text{inv} \land (\forall o \mid o \in p.\text{deps} \Rightarrow \neg o.\text{inv} \lor \mathcal{I}_C(o)[E/p.f]) \}
\]
\( p.f := E \)
Friendship discipline

**Ghost field:** \( \text{deps} : \text{set of} \ \text{Object} := \emptyset \)

**Obligation (absence of interference, as a precondition):**
\[
\neg p.\text{inv} \land (\forall o \mid o \in p.\text{deps} \Rightarrow \neg o.\text{inv} \lor \mathcal{I}_C(o)[E/p.f])
\]

\( p.f := E \)

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

**Ghost field:**  \( \text{deps} : \text{set of} \ Object := \emptyset \)

**Obligation (absence of interference, as a precondition):**
\[
\neg p.\text{inv} \land (\forall o \mid o \in p.\text{deps} \Rightarrow \neg o.\text{inv} \lor I_C(o)[E/p.f])
\]
\[ p.f := E \]

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

**Abstract** \( I_C(o)[E/p.f] \) as declared “update guard” \( U(o, p, E) \).

**Obligation:** \( \{ I_C(\text{self}) \land U(\text{self}, g, \text{val})\} \text{self.g.f} := \text{val} \{ I_C(\text{self})\} \)
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]:

[Diagram showing subject and list with arrows and interactions]
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
- correctness 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; }
}