Verification of Object-oriented Programs
Lecture 2: from friendship to dynamic frames

Dave Naumann
Department of Computer Science
Stevens Institute of Technology

Ph.D. Fall School on Logics and Semantics of State
IT University, Copenhagen, October 2008

Partially supported by US NSF grants CNS-0627338 and CRI-0708330.
Outline of lecture

Visibility-based invariants

The friendship discipline

Variations

Dynamic frames
\( \forall p \in NS. next^* \mid (p.next = \text{null} \lor p.next.prev = p) \land \ldots \)

Invariant of *Solver*, owning all the nodes?

Can we avoid reachability by decentralization?

```java
class DNode { next, prev: DNode; invar next = null \lor next.prev = self;
```
\[ \forall p \in NS. next^* | (p. next = null \lor p. next. prev = p) \land \ldots \]

Invariant of \textit{Solver}, owning all the nodes?
Can we avoid reachability by decentralization?

\textbf{class} DNode { next, prev: DNode; \textit{invar} next = null \lor next.prev = self...}
class DNode {
    next, prev: DNode;

    invar next = null ∨ next.prev = self;

    meth m(DNode b) requires b ≠ null {
        b.next := null; b.prev := self; next := b; }

    Preserves invariant for both self and b.
Visibility-based invariants

```java
class BluNode {
    next, prev: PinkNode;
    invar next = null ∨ next.prev = self;
    ...

class PinkNode {
    next, prev: BluNode;
    invar next = null ∨ next.prev = self;

    meth m(BluNode b) requires b ≠ null {
        b.next:= null; b.prev:= self; next:= b; }
}
```

Preserves PinkNode invariant for self and BluNode invariant for b.

At-risk invariants must be in scope [Müller].

Do we have to check “all” invariants at every field update?
Visibility-based invariants

```java
class BluNode {
    next, prev: PinkNode;
    invar next = null \lor next.prev = self;
    ...

class PinkNode {
    next, prev: BluNode;
    invar next = null \lor next.prev = self;

    meth m(BluNode b) requires b \neq null {
        b.next:= null; b.prev:= self; next:= b; }
    ...
}
```

Preserves PinkNode invariant for self and BluNode invariant for b. At-risk invariants must be in scope [Müller].

Do we have to check “all” invariants at every field update?
Visibility-based invariants

class BluNode {
    next, prev: PinkNode;
    invar next = null ⊕ next.prev = self;
    ...

class PinkNode {
    next, prev: BluNode;
    invar next = null ⊕ next.prev = self;

    meth m(BluNode b) requires b ≠ null {
        b.next:= null; b.prev:= self; next:= b; } ... }

Preserves PinkNode invariant for self and BluNode invariant for b.
At-risk invariants must be in scope [Müller].

Do we have to check “all” invariants at every field update?
Recall inv/own discipline: \( \text{assert } \neg x.\text{inv}; \ x.f := E \)

to protect invariant of the form \( x.\text{inv} \Rightarrow J(x) \), where \( J \) allowed to depend on objects transitively owned by \( x \).

How ensure non-owning dependants of \( x.f \) are unpacked?

- don’t — and don’t make invariant visible.
- Track dependents in ghost state, and
- expose transition constraint that ensures preservation of their (non-exposed) invariant.

Rely-guarantee flavor.
Friendship to the rescue

Recall inv/own discipline: \texttt{assert }\neg x.\text{inv}; \ x.f := E\texttt{ to protect invariant of the form }x.\text{inv} \Rightarrow J(x), \text{ where } J \text{ allowed to depend on objects transitively owned by } x.

How ensure non-owning dependants of \texttt{x.f} are unpacked?

- don’t — and don’t make invariant visible.
- Track dependents in ghost state, and
- expose \texttt{transition constraint} that ensures preservation of their (non-exposed) invariant.

Rely-guarantee flavor.
class Master {
    time: int; invar 0 ≤ self.time;
    friend Clock reads time;

    Master() ensures inv ∧ !com;
    { time := 0 ; pack self; }

    Tick(n: int)
        requires inv ∧ !com ∧ 0 ≤ n;
        ensures time ≥ old(time);
    { unpack self; time := time + n; pack self; }

    Connect(c: Clock)
        requires inv;
        ensures c ∈ self.deps ;
    { unpack self; attach c; pack self; }
}
class Master {
    time: int; invar 0 ≤ self.time;
friend Clock reads time;
Master() ensures inv ∧ !com;
{ time := 0; pack self; }

Tick(n: int)
  requires inv ∧ !com ∧ 0 ≤ n;
  ensures time ≥ old(time);
{ expose(self){
    time := time + n; }

Connect(c: Clock)
  requires inv;
  ensures c ∈ self.deps ;
{ expose(self){attach c;}}

}

class Clock {
    t: int; m: Master ;
invar (m ≠ null ∨ self ∈ m.deps)
    ∧ 0 ≤ t ≤ m.time;
guard(∀α, β) α.time := β by α.time ≤ β;

Clock(mast: Master)
  requires mast ≠ null ∧ mast.inv;
  ensures inv ∧ ¬com;
{ m:=mast; t:=0;
    m.Connect(self); pack self; Sync(); }

Sync()
  requires inv ∧ ¬com;
  ensures t = m.time ;
{ expose(self){ t := m.time; } }
}
Friendship discipline

In Clock:
$J_{Clock}(\text{self})$ depends on $\text{self}.m.time$, does not own $m$ (the pivot)

Admissible friendship invariant: may depend on $\alpha.f$ if invariant implies $\text{self} \in \alpha.deps$ and $\alpha$’s type is a friend that exports $f$.

Proof obligation for guard $\alpha.time:=\beta$ by $\alpha.time \leq \beta$ is that

$J_{Clock} \land \underline{\alpha.time \leq \beta} \Rightarrow J_{Clock}/\alpha.time \rightarrow \beta$

In Master (which exports time to friend Clock):
Proof obligation for $\text{self}.time := \text{self}.time + n$; is

$\forall c \mid c \in \text{self.deps} \Rightarrow c.time \leq \text{self}.time + n$

and $\neg \text{self.inv}$ as before.
Friendship discipline

In Clock:
$J_{\text{Clock}}(\text{self})$ depends on $\text{self.m.time}$, does not own $m$ (the pivot)

Admissible friendship invariant: may depend on $\alpha.f$ if invariant implies $\text{self} \in \alpha.deps$ and $\alpha$’s type is a friend that exports $f$.

Proof obligation for guard $\alpha.\text{time}:=\beta$ by $\alpha.\text{time} \leq \beta$ is that
$J_{\text{Clock}} \land \alpha.\text{time} \leq \beta \implies J_{\text{Clock}} / \alpha.\text{time} \rightarrow \beta$

In Master (which exports time to friend Clock):
Proof obligation for $\text{self.time} := \text{self.time} + n$; is
$\forall c \mid c \in \text{self.deps} \Rightarrow c.\text{time} \leq \text{self.time} + n$
and $\neg \text{self.inv}$ as before.
Friendship discipline

In Clock:
\( J_{Clock}(\text{self}) \) depends on \texttt{self.m.time}, does not own \texttt{m} (the pivot)

Admissible friendship invariant: may depend on \( \alpha.f \) if invariant implies \texttt{self} \( \in \alpha.deps \) and \( \alpha \)'s type is a friend that exports \( f \).

Proof obligation for guard \( \alpha.time := \beta \) by \( \alpha.time \leq \beta \) is that
\[
J_{Clock} \land \alpha.time \leq \beta \Rightarrow J_{Clock}/\alpha.time \rightarrow \beta
\]

In Master (which exports time to friend Clock):
Proof obligation for \texttt{self.time := self.time + n}; is
\[
\forall c \mid c \in \texttt{self.deps} \Rightarrow c.time \leq \texttt{self.time + n}
\]
and \( \neg \texttt{self.inv} \) as before.
Remarks on friendship

Dependency of friend \((\text{Clock})\) on a field of granter \((\text{Master})\) is visible to granter \((\text{but} \ J\text{Clock} \text{ need not be}).\)

Friend declares guard condition under which update maintains \(J\text{Clock}\) so friends need not be unpacked.

Granter maintains \((\text{in ghost field} \ deps)\) references to dependent friends. \((\text{Compare} \ own.\)\)
Observer example

Subject has a version number that is incremented each time it is updated, and *notify* brings the View back in sync. The View maintains a copy of the state of the Subject, with its version number, in its Cache object.

Listener and View share Cache object —cf. Iterators, later.
Observer example

Subject has a version number that is incremented each time it is updated, and *notify* brings the View back in sync. The View maintains a copy of the state of the Subject, with its version number, in its Cache object. Listener and View share Cache object —cf. Iterators, later.
Formal details, friend $L$ of granter $K$

$L^L(\text{self})$ allowed to depend on $x.f$ where
- $x$ in the scope of $\forall x : K \mid \text{self} = x.\text{own} \Rightarrow \ldots$, or
- $x$ in the scope of $\forall x : K \mid \text{self} \in x.\text{deps} \Rightarrow \ldots$ and $f$ declared in class $K$ to be visible to $L$

Friendly dependence on $\text{self.g.deps}$ and $\text{self.g.inv}$ also allowed.

Stipulated precondition for $x.f := E$ with $x : L$ is
$\neg x.inv$ and $\forall p : L \mid p \in x.\text{deps} \land f \in \text{reads}_{L,K} \Rightarrow U(p, x, f, E)$
where $f \in \text{reads}_{L,K}$ means granter $K$ exports $f$ to $L$
$U(p, x, f, E)$ means the update guard holds

Oblig.: $\forall x, p \mid J^L(p) \land U(p, x, f, \nu) \Rightarrow J^L(p)/x.f\rightarrow\nu$ (sem. subst.)

Pack/unpack unchanged

Attach/detach precondition $\neg \text{self.inv}$; assign to $\text{self.deps}$
Formal details, friend $L$ of granter $K$

$J^L(self)$ allowed to depend on $x.f$ where

- $x$ in the scope of $\forall x: K \mid self = x.own \Rightarrow \ldots$, or
- $x$ in the scope of $\forall x: K \mid self \in x.deps \Rightarrow \ldots$ and $f$ declared in class $K$ to be visible to $L$

Friendly dependence on $self.g.deps$ and $self.g.inv$ also allowed.

Stipulated precondition for $x.f := E$ with $x : L$ is

$\neg x.inv$ and $\forall p : L \mid p \in x.deps \land f \in reads_{L,K} \Rightarrow U(p, x, f, E)$

where $f \in reads_{L,K}$ means granter $K$ exports $f$ to $L$

$U(p, x, f, E)$ means the update guard holds

Oblig.: $\forall x, p \mid J^L(p) \land U(p, x, f, v) \Rightarrow J^L(p)/x.f \rightarrow v$ (sem. subst.)

Pack/unpack unchanged

Attach/detach precondition $\neg self.inv$; assign to $self.deps$
Formal details, friend \( L \) of granter \( K \)

\( J^L(\text{self}) \) allowed to depend on \( x.f \) where

- \( x \) in the scope of \( \forall x : K \mid \text{self} = x.\text{own} \Rightarrow \ldots \), or
- \( x \) in the scope of \( \forall x : K \mid \text{self} \in x.\text{deps} \Rightarrow \ldots \) and \( f \) declared in class \( K \) to be visible to \( L \)

Friendly dependence on \( \text{self.g.deps} \) and \( \text{self.g.inv} \) also allowed.

Stipulated precondition for \( x.f := E \) with \( x : L \) is
\[ \neg x.\text{inv} \text{ and } \forall p : L \mid p \in x.\text{deps} \land f \in \text{reads}_{L,K} \Rightarrow U(p, x, f, E) \]
where \( f \in \text{reads}_{L,K} \) means granter \( K \) exports \( f \) to \( L \)
\( U(p, x, f, E) \) means the update guard holds

Oblig.: \( \forall x, p \mid J^L(p) \land U(p, x, f, v) \Rightarrow J^L(p)/x.f \rightarrow v \) (sem. subst.)

Pack/unpack unchanged

Attach/detach precondition \( \neg \text{self.inv} \); assign to \( \text{self.deps} \)
Iterator and Collection with their own invariants

```java
class Collection {
    ver: int; state: State;
    friend Enumerator reads ver, state;
    invar ∀o : Enumerator | o ∈ self.deps ⇒
        (o.vsn ≤ ver ∧ o.coll = self ∧ self ∈ o.deps);

    guard α.vsn := β by β ≤ ver; (∀α, β)
    guard α.coll := β by β = self;

    GetEnumerator(): Enumerator
        ensures result in self.deps;
    { result:= new Enumerator(self); }

    Add(i: int) requires inv ∧ !com;
    { expose(self){ ver:= ver+1; state:= ...; } }
}
```
class Enumerator { vsn: int; coll: Collection;
    model elts: State
friend Collection reads vsn, coll;
invar vsn \leq coll.ver \land self in coll.deps \land 
    (vsn = coll.ver \Rightarrow elts.Equals(coll.state));
guard coll.ver := \beta \ by \ coll.ver \leq \beta;
guard coll.state := \beta \ by \ coll.ver \neq vsn;

Enumerator(c : Collection)
    requires c.inv \land !c.com;
    ensures self in c.deps \land coll=c \land c insel.deps;
{ vsn:= c.ver; coll:= c; attach c; c.Attach(self);
    elts:= c.state; pack self; }

MoveNext(): bool requires vsn = coll.ver { ... }
}
History constraints

History constraint: two-state “invariant” imposed on every pair of successive states (or directly successive). E.g., version number of a Subject increases monotonically.

Instead of Friend exporting a constraint about updates that preserve its invariant, the Granter can export a constraint on what updates it will do. [N,Barnett],[Leino, Schulte]
Immutability

Frozen object: transfer ownership to the “freezer”, whereupon effectively immutable. [Leino, Manohan’08]
Creational invariants

Falsifiable by allocation: \( \forall s : \text{Singleton} \mid s = \text{Singleton.theOnly} \)

Explicitly disallowed in friendship invariants.

Can allow, using “creation guards” [Pierik, Clarke, deBoer]
Dynamic frames

Ownership and friendship:

- ghost state that encodes the footprint of an object invariant (or other predicate)
- \( o = p.own \) means \( J(o) \) may depend on \( p.f \)
  \( o \in p.deps \) means \( J(o) \) may depend on \( p.f \)
- proof obligations to prevent commands from falsifying the predicate
- achieve localized reasoning about challenging patterns
- justified by meta-theorems

Kassios’ dynamic frames: programmer defined ghost footprints, for invariants and other framing done explicitly in assertions. \( o \in p.rep \) means \( J(p) \) may depend on \( o.f \)
Dynamic frames

Ownership and friendship:

- ghost state that encodes the footprint of an object invariant (or other predicate)
- $o = p.own$ means $\mathcal{I}(o)$ may depend on $p.f$
- $o \in p.deps$ means $\mathcal{I}(o)$ may depend on $p.f$
- proof obligations to prevent commands from falsifying the predicate
- achieve localized reasoning about challenging patterns
- justified by meta-theorems

Kassios’ dynamic frames: programmer defined ghost footprints, for invariants and other framing done explicitly in assertions.
- $o \in p.rep$ means $\mathcal{I}(p)$ may depend on $o.f$
Dynamic frames using pure methods

class ArrayList { count: int; items: Object[];
  ArrayList() ensures invar() \( \land \) size() = 0 \( \land \) footprint().isFresh();
  add(Object o) {
    requires invar() ensures invar() \( \land \) ... footprint().newElemsFresh();
    effects wr footprint()

  pure get(i: int): Object requires invar() \( \land \) 0 \( \leq \) i \( \leq \) size()
    effects rd footprint()

  pure footprint(): set of locations
    requires invar() effects rd footprint() // self-framing
    { return {\&count, \&items} \( \cup \) elems(items) }

  pure invar(): bool...
  }

class Stack { al: ArrayList;
  pure footprint(): set of locations {return {\&al} \( \cup \) al.footprint() }
  ...

Exercises

Do John’s exercises!
Summary

Early work on ownership: attempts at confinement notion that allows some design patterns, informal hope of modular reasoning.

Boogie: several variations, still verbose, intricate axiomatics difficult to compare with, e.g., separation logic.

Friendship: handles difficult patterns but yet more intricacy.

Pierik’s diss: creation guards, soundness and completeness for a proof-outline logic. Prototype verifier including friendship etc.

Kassios’ diss: dynamic frames (effects refer to ghost state)
References

[Müller et al’06] Modular invariants for layered object structures

[Naumann, Barnett’06] Towards imperative modules: reasoning about invariants and sharing of mutable state

[Kassios’06] Dynamic framing: support for framing, dependencies and sharing without restriction

[Smans et al’08] An automatic verifier for Java-like programs based on dynamic frames

[Pierik’06] Validation techniques for object-oriented proof outlines (Dissertation, Universiteit Utrecht)