Lecture 3

Towards a Verifying Compiler: Verifying Object Invariants

Wolfram Schulte
Microsoft Research

Formal Methods 2006

Program Invariants, Callbacks,
Aggregates, Ownership, Visibility

Joint work with **Rustan Leino, Mike Barnett**, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and *Peter Müller* (ETH), *Bart Jacobs* (KU Leuven) and Bor-Yuh Evan Chung (Berkley)

Review: Verification of OO Programs

- What is needed for designing a verifier?
- Which programs can we verify?
- What are the limitations?

Pre- and Postconditions are not Enough

Contracts can break abstraction

```
class C{
      private int a, z;
      public void M()
         requires a!=0;
         {z = 100/a;}
}
```

We need invariants

```
class C{
      private int a, z;
      invariant a!=0;
      public void M()
        {z = 100/a;}
}
```

Dealing with Invariants

- *Basic Methodology*
- Object-based Ownership
- Object-oriented Ownership
- Visibility based Ownership

Problem: Reentrancy

```
class Meeting {
  int day;  int time;
  invariant 0 ≤ day<7 ∧
     day==6 ⇒ 1200<= time;

  void Reschedule(int d )
    requires 0 ≤ d < 7;
  {
    day = d;
    X.P(this);
    if ( day==6 )  time = 1200;
  }
}
```

How can we prevent that current object is re-entered in an inconsistent state?

Program Model for Object Invariants

- Objects can be *valid* or *mutable*
  - inv $\in$ { valid, mutable } is a new ghost field in each object
- Mutable objects need not satisfy their invariants
- o.inv indicates whether the *invariant* of o, Inv(o), is *allowed to be broken*

$$\boxed{\forall o:\ o.inv \neq mutable \Rightarrow Inv(o)}$$

Remark: Quantifier ranges over allocated, non-null objects

Field Updates

- Only fields of mutable objects can be updated

```
Tr[[o.f = e]] =
    assert o≠null ∧ o.inv=mutable; o.f := e
```

Pack and Unpack

inv is changed by special source commands

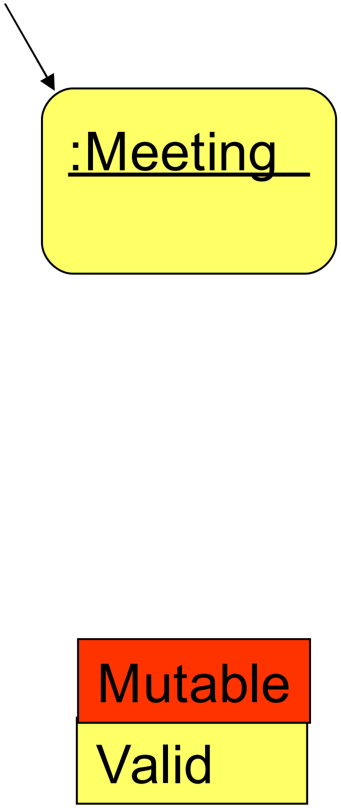
- *unpack*(o) to make o mutable
- *pack*(o) to re-establish invariant and make o valid

```
Tr[[unpack o]] =
    assert o.inv = valid;
    o.inv := mutable
```

```
Tr[[pack o]] =
    assert o.inv = mutable;
    assert Inv(o);
    o.inv := valid
```

Pack and Unpack Example

```
void Reschedule( int d )
   requires inv==valid ∧ 0≤
    d<7;
 {
   expose(this){
    day = d;
    x.P(this);

    if ( day==6 )  time = 1200;
   }
 }
```



Spec# uses expose, defined by
expose(o) s; = unpack o; s; pack o;

Program Invariant

- Theorem (Soundness)

  $$\boxed{\forall o:\ o.inv \neq mutable \Rightarrow Inv(\ o\ )}$$

- Admissible invariants contain only field accesses of the form this.f

- Proof sketch
  - new:
    new object is initially mutable
  - o.f := E;
    can only affect invariant of o, asserts o.inv = mutable
  - unpack(o):
    changes o.inv to mutable
  - pack(o):
    asserts Inv(o)

Dealing with Invariants

- Basic Methodology
- *Object-based Ownership*
- Object-oriented Ownership
- Visibility based Ownership

Problem: Object Structures

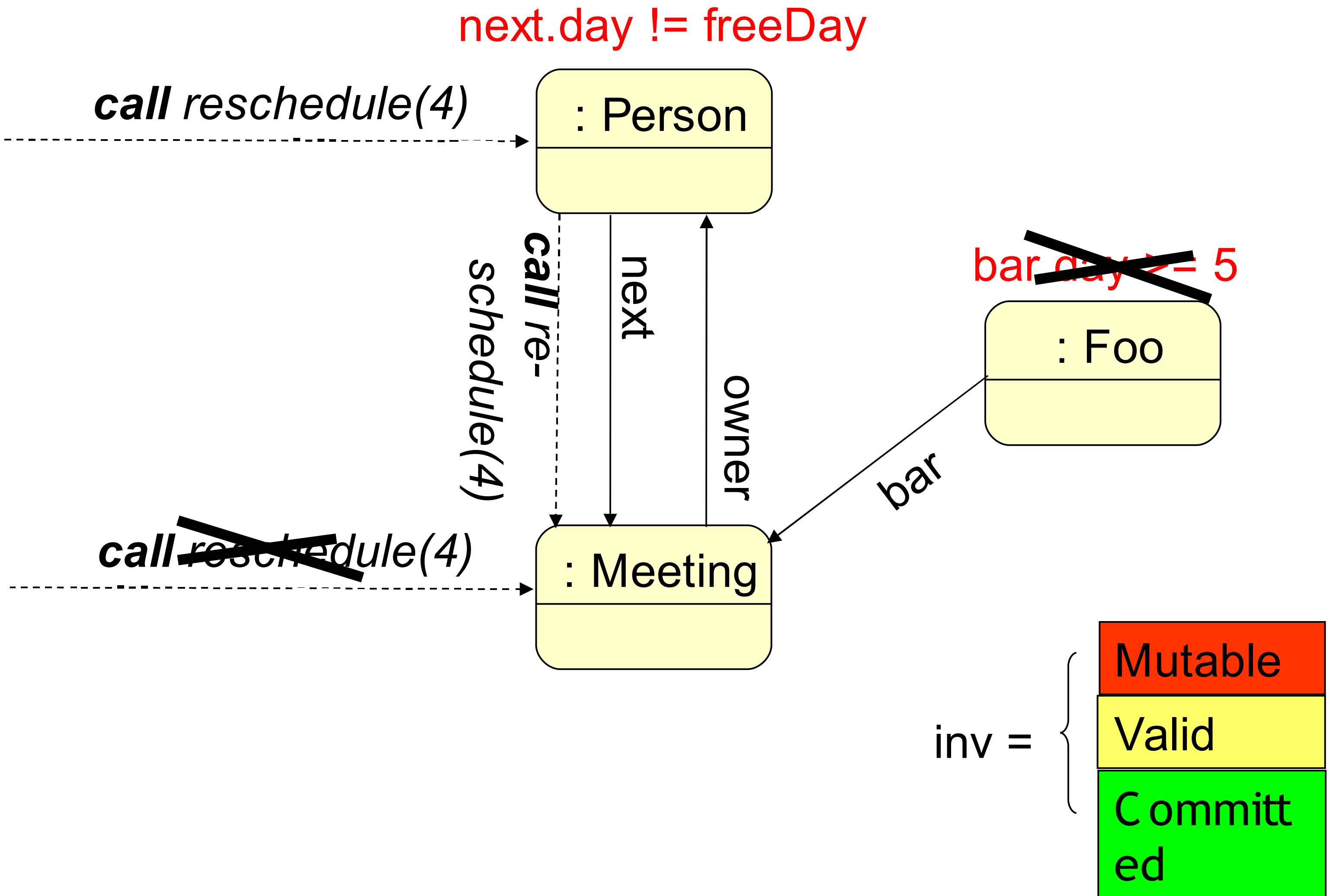
```
class Meeting {
  int day;
  invariant 0 ≤ day<7;

  void Reschedule(int d )
    requires inv==valid;
  {
    expose(this){
      day = d;
    }
  }
}
```

```
class Person {
  int freeDay;
  Meeting next;
  invariant this.next != null ⇒
      this.next.day != freeDay;
}
```

Can we have relax the admissabilty condition?

How can we find out that reschedule might break Person’s invariant?

Invariants in the Presence of Aliasing?



next.day != freeDay

call *reschedule(4)*

: Person

call *re-schedule(4)*

next

owner

~~bar.day >= 5~~

: Foo

bar

call *~~reschedule~~(4)*

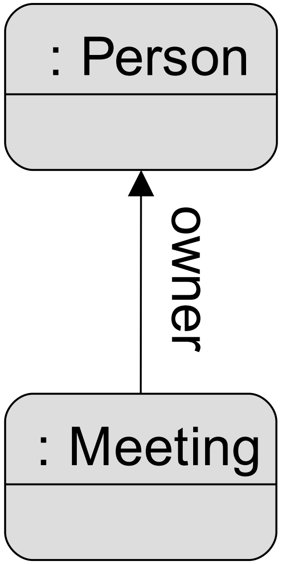
: Meeting

inv =

- Mutable
- Valid
- Committed

Ownership-Based Invariants

- Establish hierarchy (*ownership*) on objects
- *Ownership rule:* When an object is mutable, so are its (transitive) owners
- An object o may only depend on
  - the fields of o and
  - the fields of objects (transitively) owned by o



Dynamic Ownership

- Each object has a special ghost field, *owner*, that points to its owner object
- *rep*(resentation) declarations lead to *implicit owner invariants*
- *inv* ∈ {committed, valid, mutable}
- An object is committed, if
  - its invariant is known to hold
  - the owner is not mutable

```
class Person {
  int freeDay;
  rep Meeting next;

  /*implicit*/ invariant
      next ≠ null ⇒
            next.owner = this;
  …
}
```

Pack and Unpack with Ownership

- unpack(o) and pack(o) and change inv for o and o's rep objects

```
Tr[[unpack o]] =
    assert o.inv = valid;
    o.inv := mutable;
    foreach (c | c.owner = o)
      { c.inv := valid; }
```

```
Tr[[ pack o]] =
    assert o.inv = mutable;
    assert ∀c: c.owner = o ⇒
        c.inv = valid;
    foreach (c | c.owner = o)
      { c.inv := committed; }
    assert Inv( o );
    o.inv := valid
```

Program Invariant with Ownership

Theorem (Soundness)

$$\begin{aligned}&\forall o\colon o.\mathrm{inv} \neq \mathrm{mutable} \Rightarrow\\ &\qquad \mathrm{Inv}(o) \wedge\\ &\qquad (\forall c\colon c.\mathrm{owner} = o \Rightarrow c.\mathrm{inv} = \mathrm{committed}))\end{aligned}$$

Admissible invariants contain only field accesses of the form this.f_1. … .f_n where f_1 … .f_{n-1} must be rep fields

Method Framing Revisited

Allow methods to modify also committed objects

Example. Given

```
class A{ rep B b;}
class B{ rep C c;}
```

the method

```
static void m(A a) requires a.inv == valid; modifies a.*;
```

is allowed to modify …

the fields of *a.b* and *a.b.c*

This addresses the transitivity problem of modifies clauses

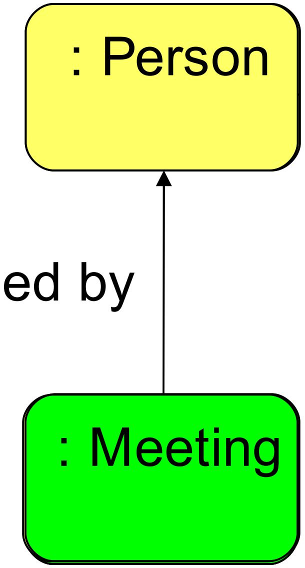
Method Framing Revisited

Allow methods to modify also committed objects

The Post condition for a Spec# modifies W clause

$$\begin{array}{l}
\mathrm{Tr}\ [[W]] = \\
(\forall o : \mathrm{ref},\ f : \mathrm{field} :: \mathrm{old}(\mathrm{Heap}[o,\mathrm{allocated}]) \\
\quad \Rightarrow (o,f) \in \mathrm{old}(W) \vee \\
\qquad \mathrm{old}(\mathrm{Heap}[o,f]) = \mathrm{Heap}[o,f]) \vee \\
\qquad \mathrm{old}(\mathrm{Heap}[o,\mathrm{inv}]) = \mathrm{committed}
\end{array}$$

Example Revisited



```
class Person {
  int freeDay;
  rep Meeting next;

  invariant next ≠ null ⇒
     next.day ≠ freeDay;


  int doTravel(int td)
    requires inv==valid; modifies
     this.*;
  { expose(this) {
     freeDay = td;
     if (next!=null) {
       next.reschedule((td+1)%7);
    };
  }
```

```
class Meeting {
  int day;

  void reschedule( int d )
    requires inv==valid;
  { expose(this)
      day = d;
    }
  }
```

Dealing with Invariants

- Basic Methodology
- Object-based Ownership
- *Object-oriented Ownership*
- Visibility based Ownership

Inheritance

- *Each subtype defines one frame* with its variables
- Single inheritance results in a sequence of frames

Example

```
class Cell { int x;
    invariant x>=0;...}
class BackupCell: Cell {
    rep History h;
    invariant h.last>=x;
    invariant x>10;
```

Objects of

- type Cell have 2 frames: [Cell, object]
- type B have 3 frames: [BackupCell, Cell, object]

- *Subtypes are allowed to strengthen invariants*

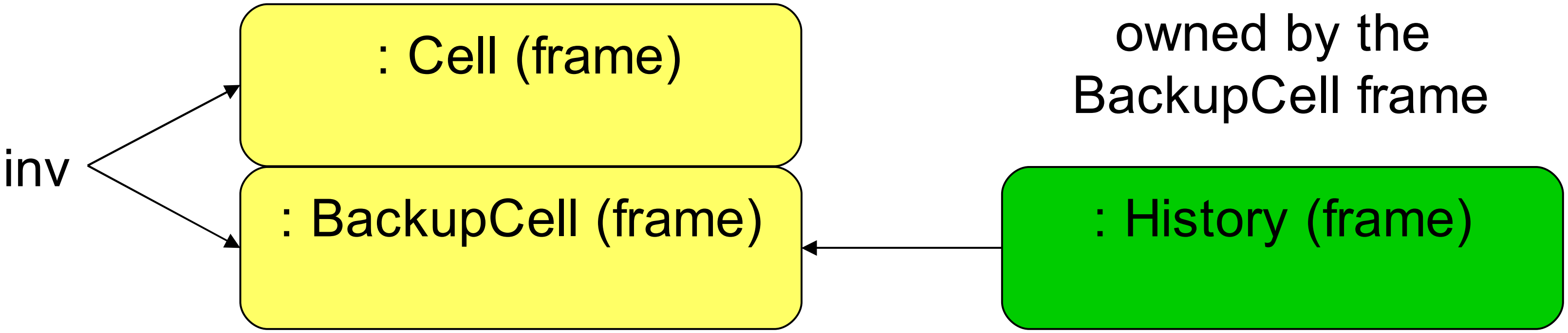
Refined Representation

Idea: Reduce inheritance to ownership of frames.

If B is a direct subtype of A, then B has a rep field A.
But we only have one inv field, so:

- o.inv now represents the most derived frame for o which is valid, i.e.
  - o.inv <: T   means Inv(o) holds for the frame T and all its super frames
  - o.inv == typeof(o) means Inv(o) holds for all of o’s frames
- o.owner is now a pair (p,T);
  - p is the owner,
  - T the frame that contains the rep field that points to o.

Refined Representation



owned by the
BackupCell frame

packed as BackupCell

packed as Cell

🟥 Mutable
🟨 Valid
🟩 Committed

Commiting c to p means then
c.commited ⇔ let (p,T) = c.owner in p.inv <: T

Refined Representation

- rep fields f in class T give rise to implicit invariants

invariant this.f!= null ⇒ let (p,T’) = this.f.owner in p== this ∧ T = T’

Pack and Unpack with Inheritance

Given class T:S and o of type T. Then

```
Tr[[unpack(o from T)]] =
    assert o.inv = T;
    assert !o.committed;
    o.inv := S
    foreach (c | c.owner==(o,T))
      { c.committed := false }
```

```
Tr[[ pack o as T]] =
    assert o.inv = S;
    assert $Inv_T$( o );
    assert $\forall$c: c.owner==(o,T)$\Rightarrow$
        c.inv = typeof(r);
    foreach (c | c.owner==(o,T))
        { c.committed := true }
    o.inv := T
```

pack(o as T) claims every object that has o as its owner and its rep field declared in T

Inheritance Precondition Problem

```
virtual void Cell.Set(int x)
   requires …
   modifies this.*;
{
    unpack(this from Cell);
       this.x = x;
    pack(this to Cell);
}
```

```
override void BackupCell .Set(int x)
  //requires …
{
   unpack(this from BackupCell);
      this.b = this.x;
     base.Set(x);
   pack(this to BackupCell);
  }
```

```
void M(Cell c)
{
   c.Set(23);
}
```

How can we verify the dynamically dispatched c.Set call?

Dynamic Dispatch and Preconditions

For virtual methods m, we allow to write the pre:

this.inv == 1

For each frame in which m is defined we generate *2 procs*

- *m.C* is used for *statically bound calls*; its pre:

  Heap[o,inv] = C

- *m.C.Virtual* is used for *dynamically dispatched calls,* its pre:

  Heap[o,inv] = typeof(o)

Only *m.C* contains the translated code

Inheritance Precondition Example

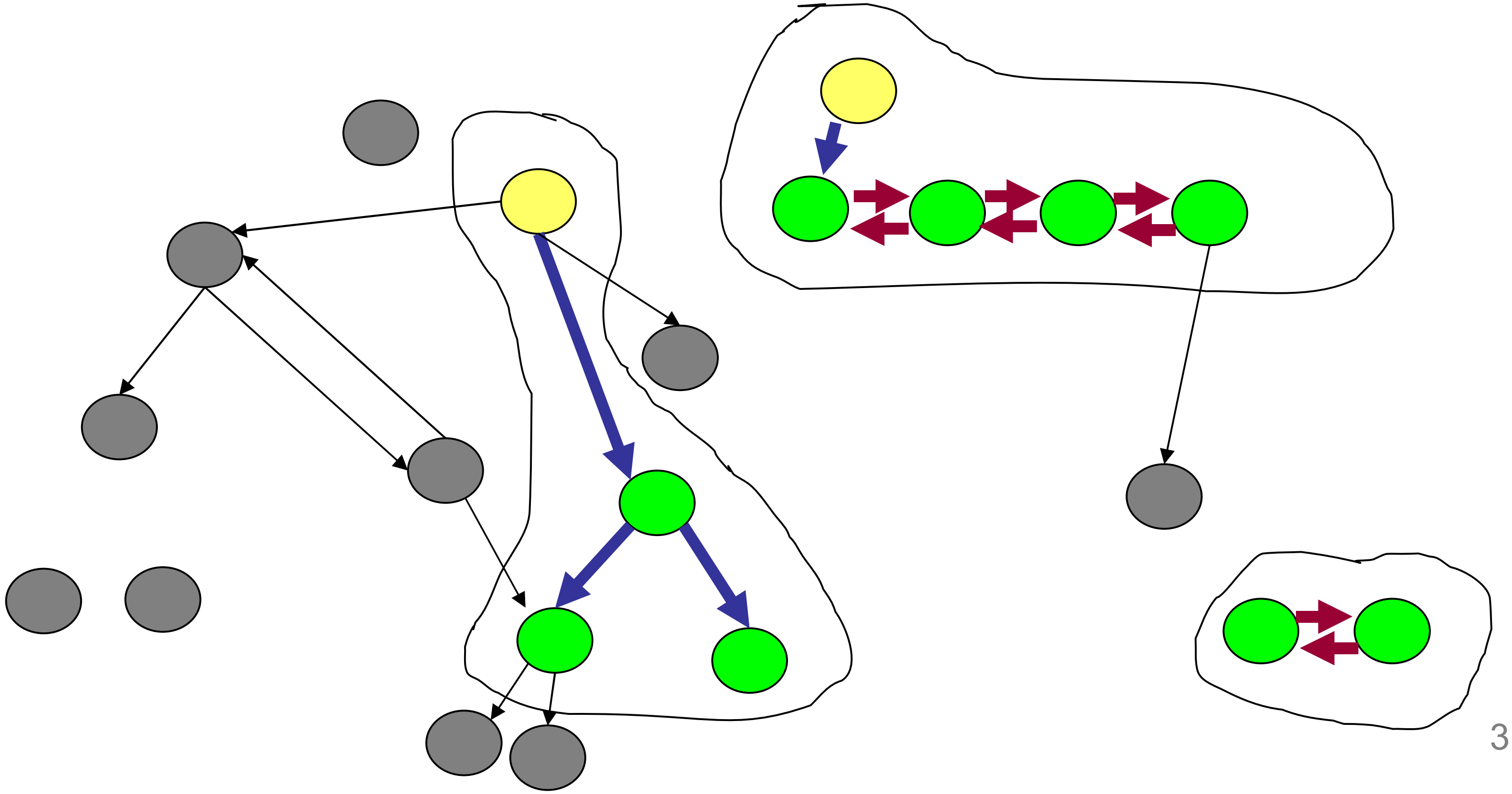
```
virtual void Cell.Set(int x)
   requires this.inv == 1;
   modifies this.*;
{
   unpack(this from Cell);
        this.x = x;
   pack(this to Cell);
}



void M(Cell c)
  requires c.inv == typeof(Cell);
{
  c.Set(23);
}
```

Dealing with Invariants

- Basic Methodology
- Object-based Ownership
- Object-oriented Ownership
- *Visibility based Ownership*

Rep & Peer Objects

- *Rep fields* are used to *build hierarchical abstractions*.
- *Peer fields* are used to *group sibling objects.*



Rep and Peer Objects

```
class List {
 rep  Node! head;
 invariant
    (∀ Node n | n.owner=this ⇒
       next!=null ⇒ next.prev =
    this
      ∧…
}
class Node{
  peer Node next, prev;
}
```

```
class List {
 rep Node! head;
}
class Node{
  peer Node next, prev;
  invariant
    next!=null ⇒ next.prev = this
    ∧ …
}
```

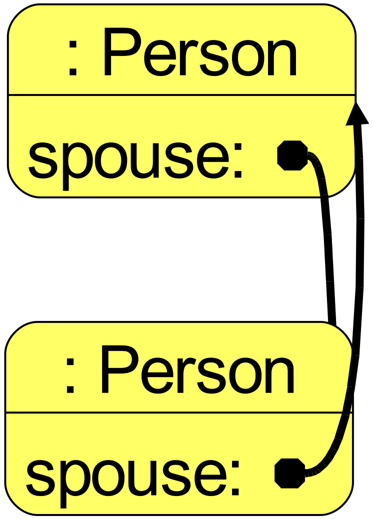
Peer fields give rise to additional invariants, for Nodes e.g.

next!=null ⇒ owner = next.owner ∧ prev!=null ⇒ owner = prev.owner

But how can we reason locally?

Mutually Recursive Object Structures

```
class Person {
  Person spouse;


  invariant this.spouse ≠ null ⇒
                this.spouse.spouse = this;
 …
}
```



- Objects are mutually dependent
- Objects cannot own each other

Admissible Visibility-Based Invariant

```
class Person {
 Person spouse dependent Person;

 invariant  this.spouse ≠ null ⇒ this.spouse.spouse = this;
 …
}
```

The invariant declared in class C may contain a field access R.f iff

- R is “**this**” or
- R is “**this**.g_1. … .g_n.” and g_1..g_n are rep or peer fields, and *C is mentioned in the dependent clause of f*

Proof Obligation for Field Updates

- For a field update o.f := E; we have to prove
  - That o is non-null and mutable (as before)
  - That all other objects whose invariant depends on o.f are mutable

```
Tr[[o.spouse := E; ]]
 assert  o ≠ null ∧ o.inv = mutable;
 assert  ∀Person t: t.spouse = o ⇒ t.inv = mutable;
```

- The other objects are determined by *inspecting the invariants* of the all “friend” classes mentioned in the dependent clause (see next slide)

Marriage is Never Easy…

```
class Person {
 Person spouse dependent Person;
 invariant this.spouse ≠ null ⇒ this.spouse.spouse = this;

 void marry( Person p )
   requires p≠null ∧ p≠this ∧ this.inv = valid ∧ p.inv = valid ∧
                    this.spouse = null ∧ p.spouse = null ;
 { expose(this)
     expose(p) {
       this.spouse := p;
       p.spouse := this;
     }
   }
}
```

this.inv = mutable ∧ this.spouse = null

p.inv = mutable ∧ p.spouse = null

this.spouse.spouse=this∧ p.spouse.spouse=p

Summary: Object Invariants

- The methodology solves the problems of
  - Re-entrance (through the explicit inv field)
  - Object structures (through ownership or visibility)
- It can handle
  - Complex object structures including (mutual) recursion
  - Ownership transfer (not shown in this talk)
- The methodology is modular and sound
- Aliasing is not restricted