

2.4 Subtyping, Inheritance, Extended State

Problems and Spec# approach:

0. Subclass methods are executed in contexts where only base class is known:
 - No refinement of preconditions in overriding methods
 - $\text{postcond:subclass} \implies \text{postcond:base_class}$
1. Inherited methods could break subclass invariants:
 - [Additive] hint to modular verifier
2. Overriding methods could break superclass invariants:
 - Discipline to stepwise expose objects
 - Class frames

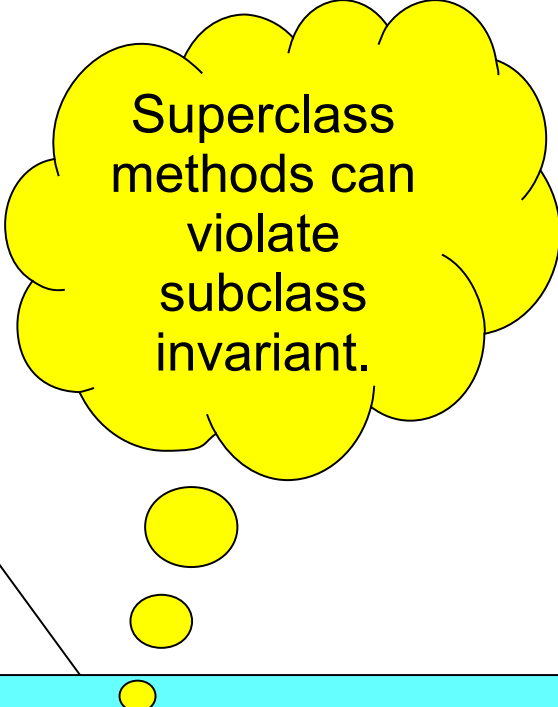
Illustration by a “speedy” example:

```
public class Car {  
    protected int speed;  
    invariant 0 <= speed;  
  
    protected Car()  
    { speed = 0; }  
  
    public void SetSpeed( int kmph )  
        requires 0 <= kmph;  
        ensures speed == kmph;  
    {  
        expose (this) { speed = kmph; }  
    }  
}
```

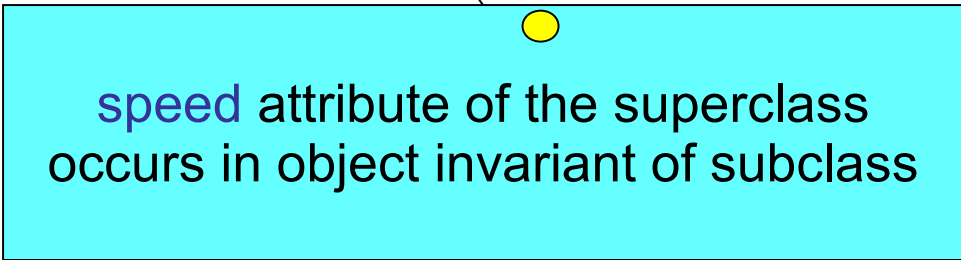
1.a Inheriting class: Additive invariants

```
public class LuxuryCar : Car
{
    int cruiseControlSetting;
    invariant cruiseControlSetting == -1 ||
               cruiseControlSetting == speed;

    LuxuryCar()
    { cruiseControlSettings = -1; }
}
```



Superclass
methods can
violate
subclass
invariant.



speed attribute of the superclass
occurs in object invariant of subclass

1.b Changes required in the base class:

```
public class Car {
    [Additive] protected int speed;
    invariant 0 <= speed;

    protected Car()
    { speed = 0; }

    [Additive]
    public void SetSpeed( int kmph )
        requires 0 <= kmph;
        ensures speed == kmph;
    {
        additive expose (this)
        { speed = kmph; }
    }
}
```

[Additive] annotation is needed
as `speed` is mentioned in
object invariant of `LuxuryCar`

Additive `expose` is needed
as the `SetSpeed` method is
inherited and so must expose
`LuxuryCar` if called on a
`LuxuryCar` Object

2.a Virtual methods

```
public class Car {  
    [Additive] protected int speed;  
    invariant 0 <= speed;  
  
    protected Car()  
    { speed = 0; }  
  
    [Additive] virtual public void SetSpeed( int kmph )  
        requires 0 <= kmph;  
        ensures speed == kmph;  
    {  
        additive expose (this)  
        { speed = kmph; }  
    }  
}
```

2.b Overriding methods:

```
public class LuxuryCar : Car {
    int cruiseControlSetting;
    invariant cruiseControlSetting == -1    ||
               cruiseControlSetting == speed;

    LuxuryCar() { cruiseControlSettings = -1; }

    [ Additive]
    override public void SetSpeed(int kmph)
        ensures cruiseControlSettings == kmph
                && cruiseControlSettings == speed;
    {
        additive expose (this) {
            cruiseControlSettings = kmph;
            base.SetSpeed( cruiseControlSettings );
        } }
}
```

Exposes class frame
of LuxuryCar

Exposes class frame
of Car

2.5 Remarks on Further Aspects

... the course concentrated on modularity and covered only parts of Spec# --

Not treated:

- Object initialization
- Non-nullness annotations
- Purity and immutability
- Recursive dependencies among peers
- Owners per class frame

Discussion:

- Automatic modular verification technique for classes
- Main goal: detect programming errors
- Full tool support
- Data abstraction
- Specification of interface types
- Power of specification language
- Module/component concept

References / Further Reading:

Rustan Leino and Peter Müller:

Object invariants in dynamic contexts

ECOOP 2004

Mike Barnett, Robert DeLine, Manuel Fähndrich, Rustan Leino, and Wolfram Schulte:

Verification of object-oriented programs with invariants

Journal of Object Technology 2004

Peter Müller, Arnd Poetzsch-Heffter, and Gary Leavens:

Modular Invariants for Layered Object Structures

Science of Computer Programming 2006

<http://www.rosemarymonahan.com/specsharp/>

3. Modular Behavioural Specification (& Verification)

Component specification:

[Szyperski: Component Software, 2nd ed., p. 41 & p.78]

Definition of component:

"A software **component** is a unit of composition with contractually **specified interfaces** and **explicit context dependencies** only. ..."

Discussion of component specification:

"The specification problems encountered in recursive re-entrant systems need to be solved in a **modular** way to cater for components. In other words, each **component** must be **independently verifiable** based on the contractual specification of the interfaces it requires and those it provides."

Central challenges:

- What is an appropriate notion of **program components**?
- How can we specify component behaviour in an **implementation-independent** way?
- How can we **modularly** prove implementations **correct**?
- How can we prove two implementations to be **equivalent**?
- When are specifications strong enough for **compositional** reasoning?

OO-program components:

- larger than a class / smaller than a complete program
- designed by the programmer
- provide a **boundary** to component **environment**
- specifiable in an **implementation-independent** way

Components at runtime:

- have local state
- may have incoming and outgoing references
- communicate with the environment in a controlled way
- allow for callbacks
- support nesting

Overview of the following:

1. The box approach to components
2. Behavioural specifications of boxes
3. Specification of multi-access boxes

Relation to Spec# / ESC Java approach:

1. Different specification goals
2. Verification technology of Spec#/ESC Java is needed to prove implementation correctness

3.1 The Box Approach to Components

General approach:

Structure the heaps into boxes with clear **boundaries** s.t.:

- Implementation of a box consists of fixed set of classes
- Types of references crossing a boundary are known
- An object belongs to exactly one box (but boxes can be nested)

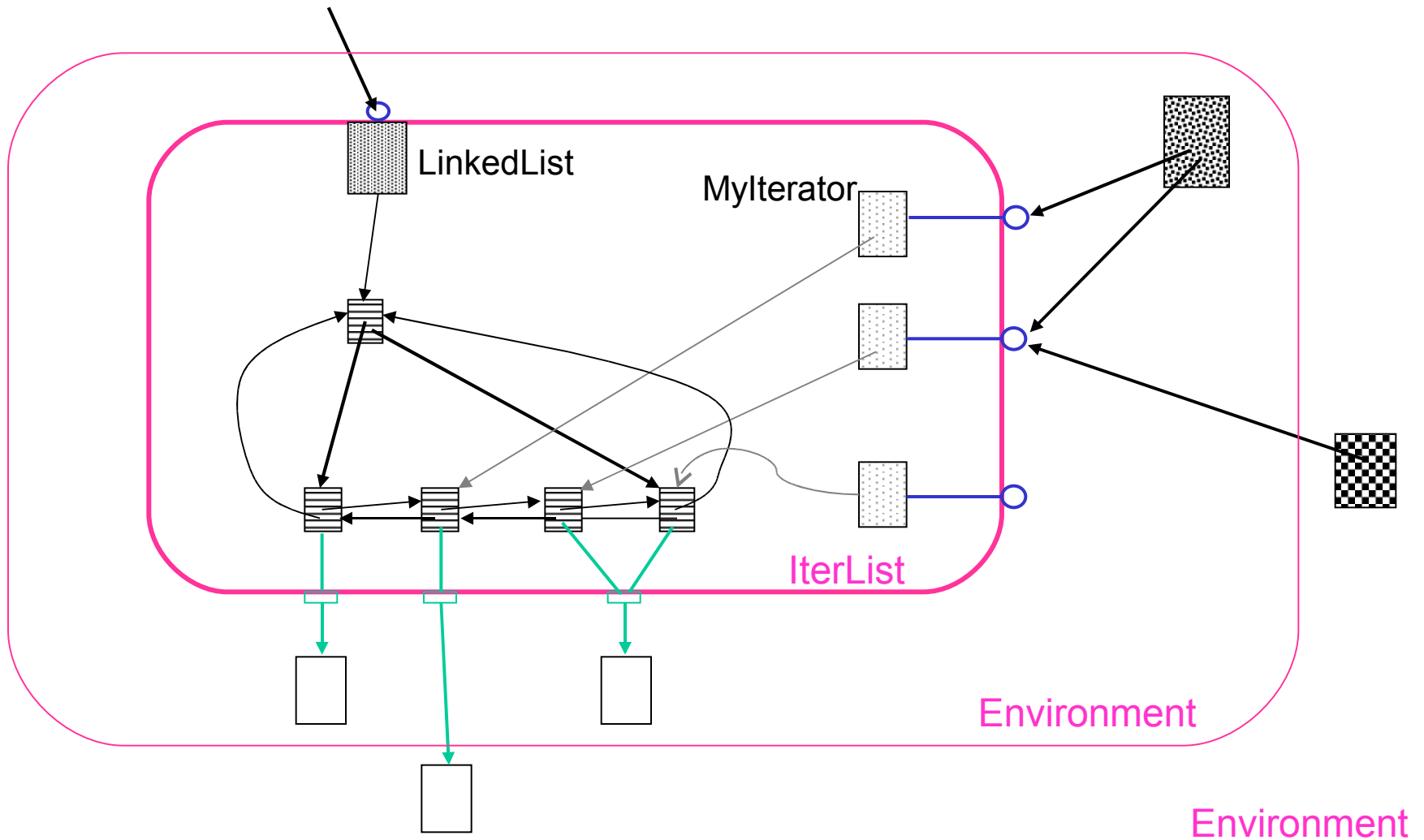
Relation to ownership techniques:

- ownership contexts with multiple ingoing references
- similar to ownership domains
- control of outgoing references

Different box disciplines:

- Each module / package P defines a box:
 - box exactly contains the objects of the classes of P
- Each module / package P defines a box:
 - box contains the objects of the classes of P and objects of classes of included packages if created from objects in the box
- Programmer defined classes create boxes at runtime:
 - box contains the newly created object X and all objects directly or indirectly created by X
- others

Boxes at runtime:



box decl IterList : IList;

```
interface IList {  
    Object getFirst() ;  
    void add( Object o );  
    Iterator iter();  
}
```

```
interface Iterator {  
    boolean hasNext() ;  
    Object next();  
    void remove();  
}
```

```
class Object {  
    boolean equals( Object y )  
    ...  
}
```

box impl IterList by LinkedList;

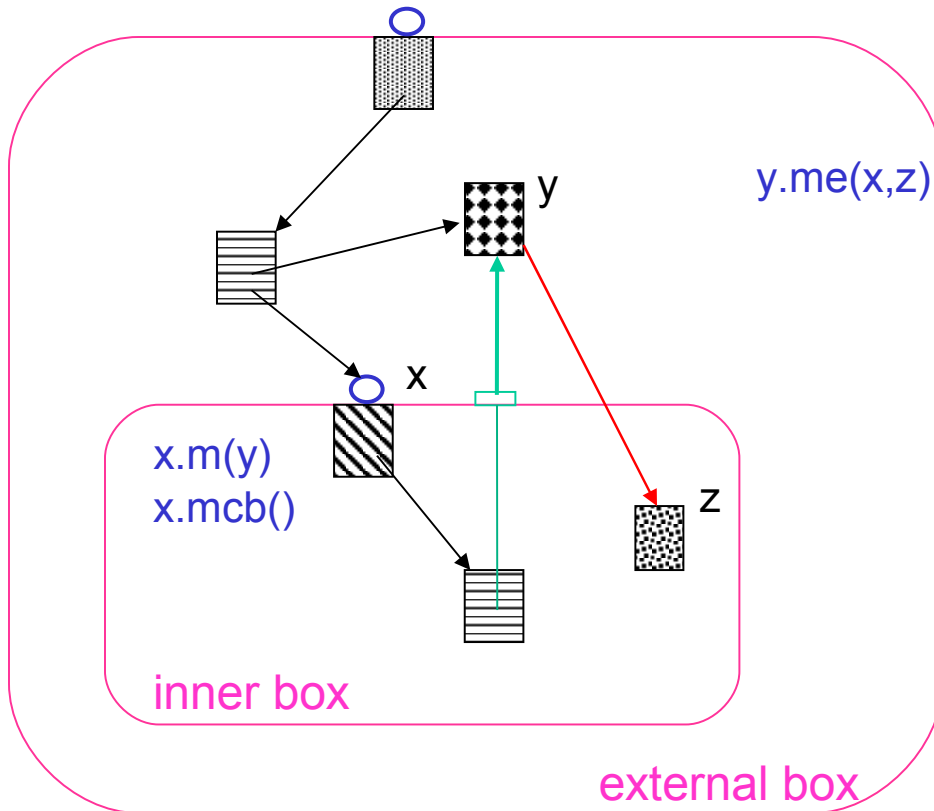
```
class LinkedList implements IList {  
    ...  
    Object getFirst() { ... }  
    void addFirst(Object o);  
    Iterator iter () {  
        return new MyIterator(this);  
    }  
}
```

```
class Entry { ... }
```

```
class MyIterator implements Iterator {  
    boolean hasNext() { .. }  
    Object next(){...}  
}
```

... **new** IterList(); ... *creates a box instance*

Dynamic behavior at box boundaries:

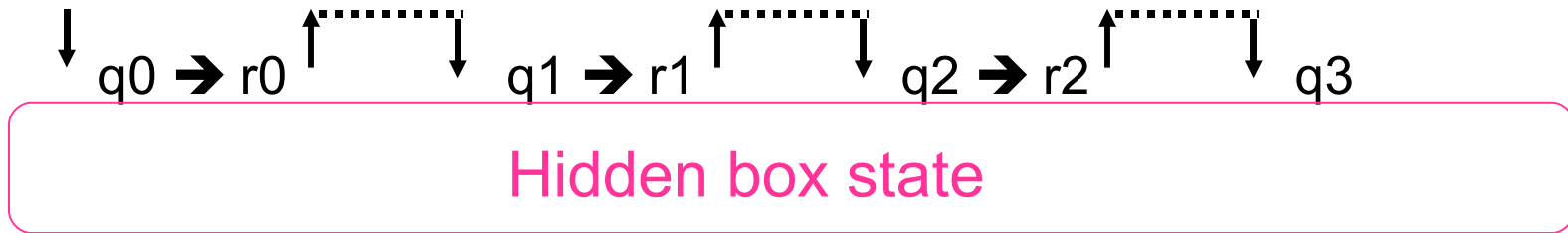


Illustrating dynamics:

- object creation
- box creation
- local state change
- boundary call
- import of references
- outgoing call
- export of references
- callbacks

3.2 Boxes: Behavioural semantics & specs

Box semantics expressed as message traces:



Specification principles:

- Specify possible traces
- States are an auxiliary means to express past history
- *Normalized* object identifiers at boundary

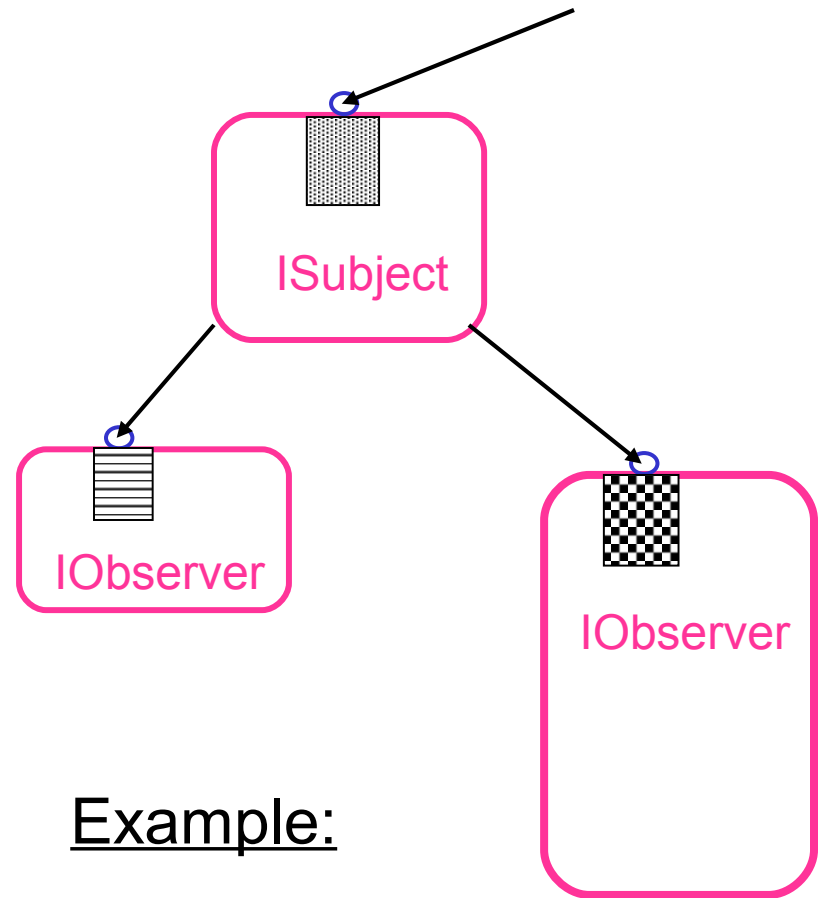
Specification technique:

- Approach:
 - Representation-independent
 - Environment-independent
 - Provides a notion of substitutability
- Constructs:
 - Model state, model programs
 - Tracking of boundary-crossing references
 - Notion of data and pure methods
- Modularity by construction:
 - Selection of box discipline (all relevant code known)
 - Well-defined boundary
 - Restriction of formulas to things controlled by box

Illustration of specification technique:

```
interface ISubject {  
    SState getS();  
    void register ( IObserver o );  
    void update( SState s );  
}
```

```
interface IObserver {  
    void notify();  
}
```



Data:

- Immutable
- Structural equality
- References are not tracked

Example:

- SState
- List<A>, Set<A>

```

box Subject implements ISubject {
  receives Set<IObserver> obs;

  model SState sbs;
  model boolean accessible;

  Subject() {
    assert obs.isEmpty() && accessible;
  }

  State getS() {
    result = sbs;
  }

  void register ( IObserver o ) {
    assume accessible ;
    obs = obs.add(o);
  }

```

Outgoing refs

Model state

Model program

Internal data call

```

...
void update( SState s )
{
  assume accessible;
  accessible = false;
  sbs = s;
  for( o in obs ){
    o.notify() ;
  }
  accessible = true;
} }

```

```
box Observer implements IObserver {  
    receives ISubject mySub;  
  
    model OState oss;  
  
    Observer( ISubject s ) {  
        s.register( this );  
        assert mySub == s;  
    }  
  
    void notify() {  
        oss = oss.succState( mySub.getState() );  
    }  
}
```

Outgoing call
in spec

Member of interface OState

Discussion:

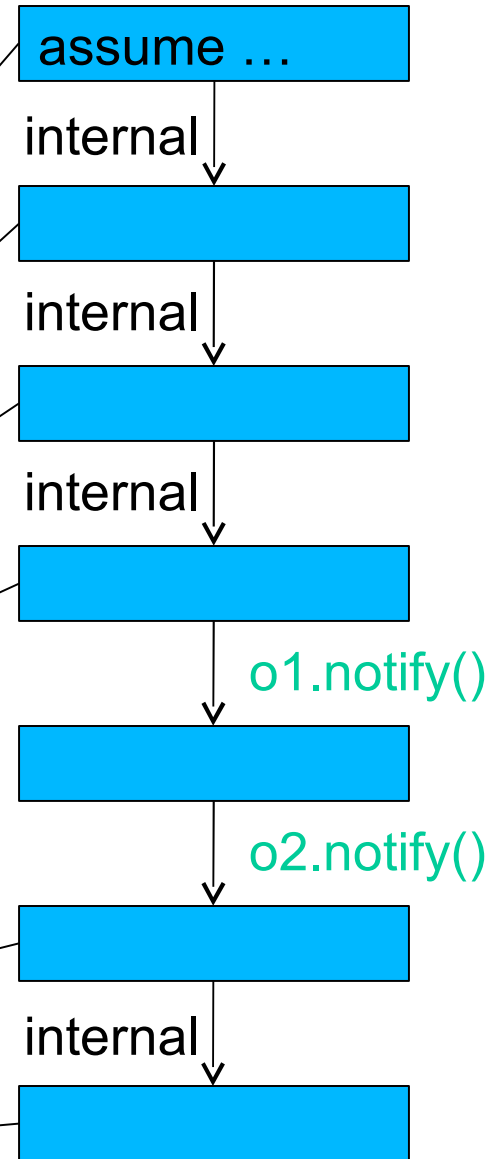
- Specifications use only local model information and signatures of used interfaces
- Spec of **Subject** makes no assumption about behaviour of **IObserver.notify()**
 - no modifies-clause for **IObserver** needed
- Specifications need no (mplementation) invariants
- Boundary invariants can be established during reasoning
- Specifications are restricted programs with a
 - trace semantics
 - reference tracking

Semantics of model programs:

Trace with:

- states
- internal actions
- external calls

```
assume accessible;  
accessible = false;  
sbs = s;  
for( o in obs )  
{  
    o.notify() ;  
}  
accessible = true;
```



Three kinds of reasoning:

1. Deriving facts from the specification:

```
State getS()  
  modifies nothing;  
{  
  result = sbs;  
  assert result == sbs;  
}
```

```
void register ( IObserver o ) {  
  assume accessible ;  
  obs = obs.add(o);  
  assert accessible && obs == pre(obs).add(o)  
           && sbs == pre(sbs);  
}
```

```
void update( State s ) {
    assume accessible ;
    accessible = false;
    sbs = s;
    for( o in obs ){
        assert !accessible && sbs == s && obs==pre(obs);
        o.notify() ;
        assume !accessible && sbs == s && obs==pre(obs);
    }
    accessible = true;
    assert accessible && sbs == s && obs==pre(obs) ;
}
```

Enriching the specification can essentially be done by

- classical program verification techniques and
- access restrictions (extended type states)

2. Using specifications to prove application programs:

Proof consistency after notification:

```
consistent: OState * SState --> bool
consistent ( os.succState( sst ), sst )
```

Application program:

```
...
State s = ...
Subject sbj = new Subject();
Observer o1 = new Observer(sbj);
Observer o2 = new Observer(sbj);
sbj.update(s);
assert consistent(o1.oss,sbj.sbs);
```

```
Subject subj = new Subject();  
assert subj.obs == {} && subj.accessible ;  
Observer o1 = new Observer(subj);  
Observer o2 = new Observer(subj);  
subj.update(s);  
assert consistent(o1.oss,subj.sbs);
```

```
Subject subj = new Subject();
assert subj.obs == {} && subj.accessible ;
subj.register(o1);
assert o1.mySub == subj ;
Observer o2 = new Observer(subj);
subj.update(s);
assert consistent(o1.oss,subj.sbs);
```

```
Subject subj = new Subject();
assert subj.obs == {} && subj.accessible ;
subj.register(o1);
assert o1.mySub == subj && subj.obs == {o1}
    && subj.accessible;
Observer o2 = new Observer(subj);
subj.update(s);
assert consistent(o1.oss,subj.sbs);
```

```
Subject subj = new Subject();
assert subj.obs == {} && subj.accessible ;
subj.register(o1);
assert o1.mySub == subj && subj.obs == {o1}
    && subj.accessible;
Observer o2 = new Observer(subj);
assert o1.mySub == subj && o2.mySub == subj
    && subj.accessible && subj.obs == {o1,o2};
subj.update(s);
assert consistent(o1.oss,subj.sbs);
```



```
Subject sbj = new Subject();
assert sbj.obs == {} && sbj.accessible ;
sbj.register(o1);
assert o1.mySub == sbj && sbj.obs == {o1}
    && sbj.accessible;
Observer o2 = new Observer(sbj);
assert o1.mySub == sbj && o2.mySub == sbj
    && sbj.accessible && sbj.obs == {o1,o2};
assume sbj.accessible;
accessible = false;
sbs = s;
o1.notify();
o2.notify();
accessible = true;
assert sbj.sbs == s && sbj.obs=={o1,o2};
assert consistent(o1.oss,sbj.sbs);
```

```
Subject sbj = new Subject();
assert sbj.obs == {} && sbj.accessible ;
sbj.register(o1);
assert o1.mySub == sbj && sbj.obs == {o1}
    && sbj.accessible;
Observer o2 = new Observer(sbj);
assert o1.mySub == sbj && o2.mySub == sbj
    && sbj.accessible && sbj.obs == {o1,o2};
o1.notify();
assert o1.oss == o1.oss.succState( sbj.sbs );
o2.notify();
assert consistent(o1.oss,sbj.sbs);
```

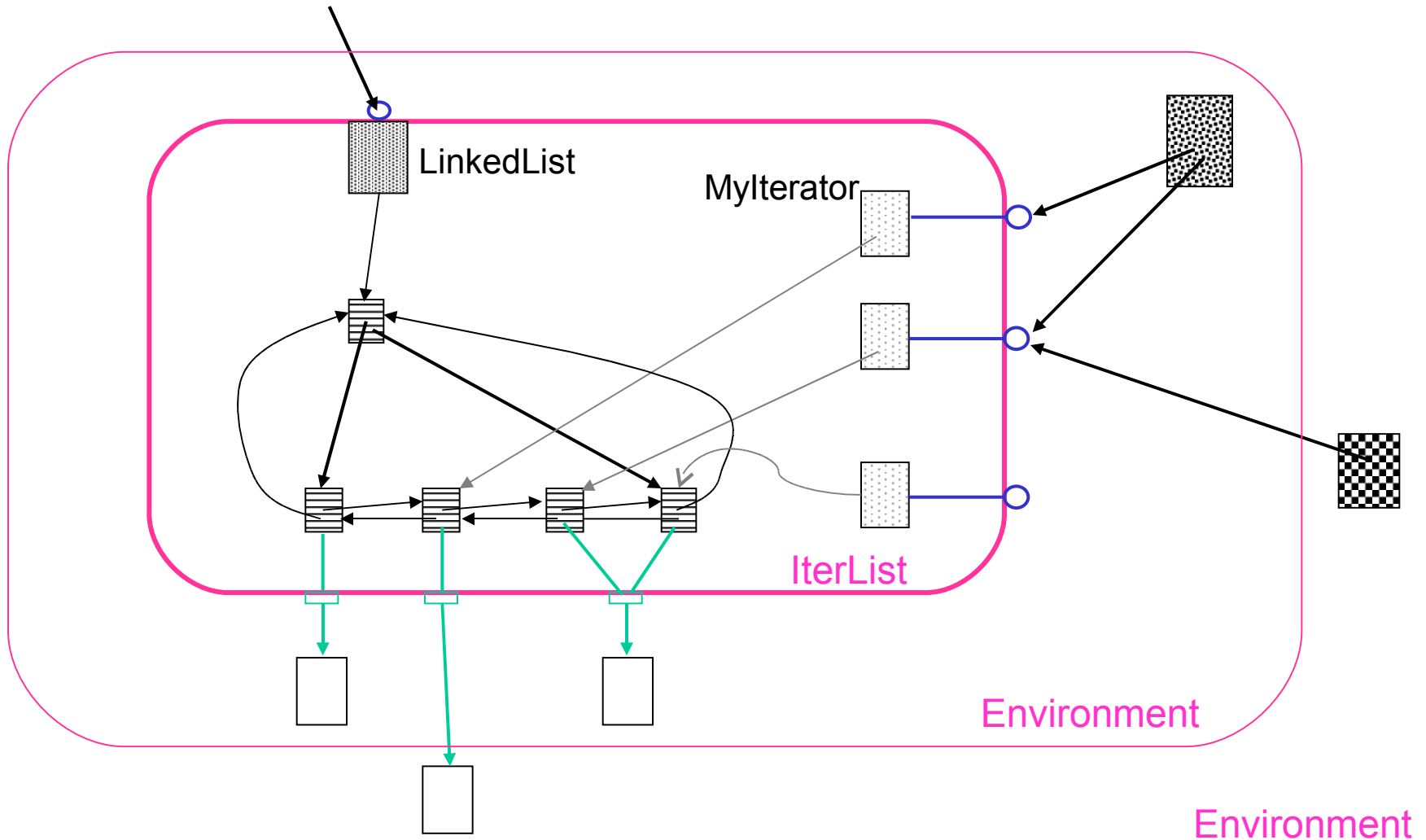
```
Subject sbj = new Subject();
assert sbj.obs == {} && sbj.accessible ;
sbj.register(o1);
assert o1.mySub == sbj && sbj.obs == {o1}
    && sbj.accessible;
Observer o2 = new Observer(sbj);
assert o1.mySub == sbj && o2.mySub == sbj
    && sbj.accessible && sbj.obs == {o1,o2};
o1.notify();
o2.notify();
assert o1.oss == o1.oss.succState( sbj.sbs );
assert consistent(o1.oss,sbj.sbs);
```

```
Subject sbj = new Subject();
assert sbj.obs == {} && sbj.accessible ;
sbj.register(o1);
assert o1.mySub == sbj && sbj.obs == {o1}
    && sbj.accessible;
Observer o2 = new Observer(sbj);
assert o1.mySub == sbj && o2.mySub == sbj
    && sbj.accessible && sbj.obs == {o1,o2};
o1.notify();
o2.notify();
assert o1.oss == o1.oss.succState( sbj.sbs )
    && consistent(o1.oss.succState( sbj.sbs ), sbj.sbs );
assert consistent(o1.oss,sbj.sbs);
```

3. Proving implementations correct:

- In general: based on refinement & simulation
- For the sketched framework: open problem

3.3 Specification of Multi-Access Boxes



The interface types:

```
interface IList {  
    Object get( int index );  
    void    add( Object x );  
    Object remove( int index );  
    boolean contains ( Object x );  
    Iterator iter();  
}
```

```
interface Iterator {  
    boolean hasNext();  
    Object next();  
    void    remove();  
}
```

The box structure:

```
box IterList implements IList {  
  receives Set<Object> elems;  
  
  model List<Object> list;  
  
  //... construcors and methods of IterList  
  
  exposes Set<Iterator> iters {  
    model IterList myList;  
    model int pos;  
    model boolean valid;  
    model boolean hasCurrent;  
  
    //... construcors and methods of Iterator  
  }  
}
```



```

IterList() {
    assert list.isEmpty() && iters.isEmpty()
        && elems.isEmpty();
}

Object get( int index ) {
    assume 0 <= index && index < list.length();
    assert result == list.nth(index);
}

void add( Object x ) {
    for( it in iters ){
        if( it.pos == list.length() ) it.valid = false;
    }
    list = list.addLast(x);
    elems = list.toSet();
}

```

```
Object remove( int index ) {  
    assume 0 <= index && index < list.length();  
    for( it in iters ){  
        if( it.pos >= index ) it.valid = false;  
    }  
    result = list.nth(index);  
    list    = list.delete(index);  
    elems  = list.toSet();  
}
```

Handling tricky OO-aspects:

```
boolean contains ( Object x ) {  
  if( x == null ) {  
    result = list.hasElem(null);  
  } else {  
    result = false;  
    for( y in list ) {  
      if( x.equals(y) ) {  
        result = true;  
        break;  
      }  
    }  
  }  
}
```

Complex (?)
outgoing call

Exposing references:

```
owned Iterator iter() {  
  assert !pre(iters).hasElem(result)  
  && iters = pre(iters).add(result)  
  && result.myList == this  
  && result.pos == 0  
  && result.valid == true  
  && result.hasCurrent == false ;  
}
```

Specification of Iterator with reuse:

```
boolean hasNext() {  
    assume valid;  
    assert result == (pos < myList.list.length());
```

```
Object next() {  
    assume valid && hasNext();  
    result = myList.get(pos);  
    pos++; hasCurrent = true;  
}
```

```
void remove() {  
    assume valid && hasCurrent  
    myList.remove(pos);  
    hasCurrent = false;  
}
```

Deriving specification invariant:

```
box IterList ... {  
  ...  
  invariant  
    forall it in iters: ( 0 <= it.pos )  
      && ( it.valid ==> it.pos <= it.myList.list.length() );  
}
```

Box invariants have to hold when control is outside the box.

Try to proof the invarinat by induction over the traces.

Discussion of the approach:

- + Modular programming language semantics
- + Clear relation to specification language semantics
- + Data and control abstraction
- Promising approach, but not yet validated in detail
- No programming logic so far
- No tool support so far

References / Further Reading:

Tony Hoare:

Proof of Correctness of Data Representations

Acta Informatica

Rustan Leino and Peter Müller:

A Verification Methodology for Model Fields, ESOP 2006

Arnd Poetzsch-Heffter and Jan Schäfer :

Modular Specification of Encapsulated Object-Oriented Components, FMCO 2005

Arnd Poetzsch-Heffter and Jan Schäfer :

A Representation-Independent Behavioral Semantics for Object-Oriented Components, FMOODS 2007

4. Concluding Remarks

- + Huge progress in program verification
- + Modularity meanwhile recognized as important
- + Software size and security/safety requirements
- + Nice tools
- Complexity of approaches
- Need to handle complex programs
- Not sufficient abstraction (data and control)
- Concurrency is still a construction site
- Area too challenging for research?