Automated Testing & Verification

ESC/Java2 vs. JMLForge

Juan Pablo Galeotti, Alessandra Gorla, Andreas Rau
Saarland University, Germany

Tools! Tools! Tools!

- ESC/Java2: the formula is built using Dijsktra's Weakes precondition. Automatic theorem prover: Simplify SMT Solver.

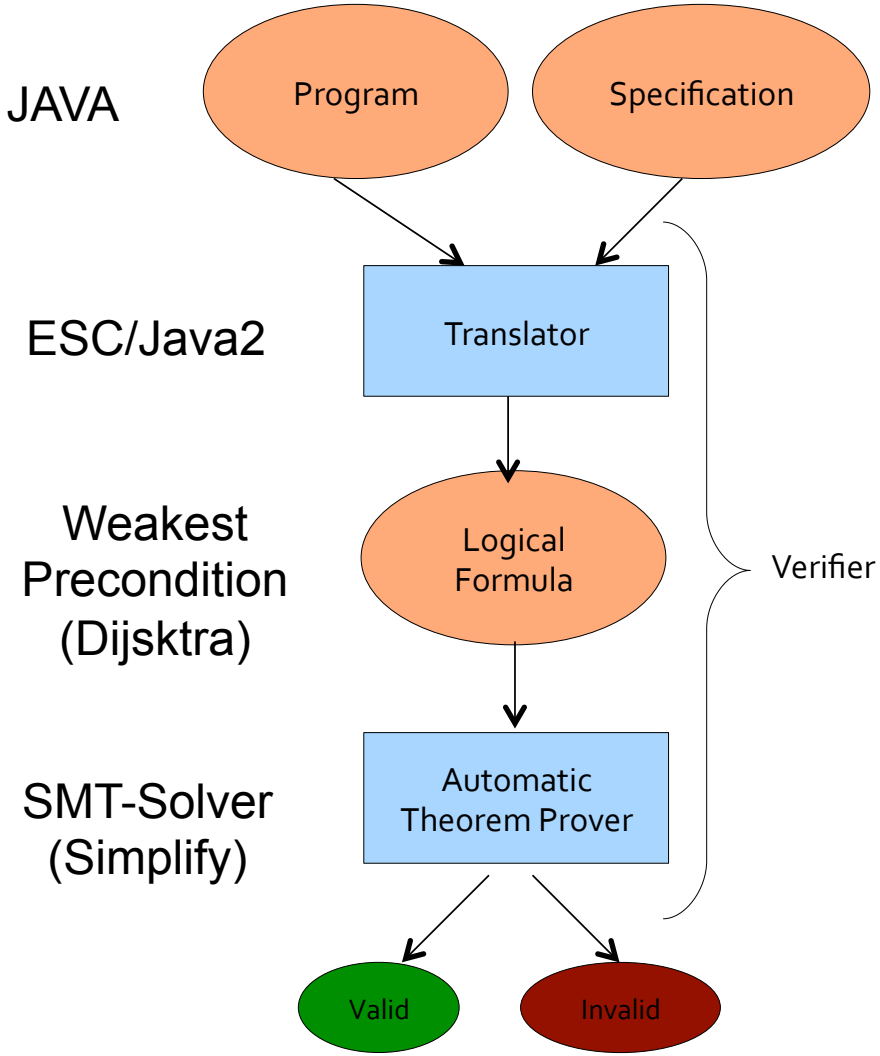
http://kindsoftware.com/products/opensource/ESCJava2/

- JMLForge: the formula is built using symbolic execution. Automatic theorem prover: off-the-shelf SAT-Solver.

http://sdg.csail.mit.edu/forge/jmlforge.html

ESC/Java2

- Programming language
- Specification Language
- Logical representation of correctness
- Automatic decision procedure



Demo ESC/Java2

```
class Bag {
        int[] a;
        int n;
        int extractMin() {
                int mindex=0;
                int m=a[mindex];
                int i=1;
                for (i=1;i<n;i++) {
                        if (a[i]<m) {
                                mindex=i;
                                m = a[i];
                        }
                }
                n--;
                a[mindex]=a[n];
                return m;
        }
```

Exercise #1

```
public class Exercise1 {

    //@ requires array!=null;
    //@ requires array.length==len;
    public int sum(int[] array, int len) {
        int sum = 0;
        int i=0;
        //@ loop_invariant i>=0;
        //@ loop_invariant i<=len;
        while (i < len) {
            sum = sum + array[i];
            i=i+1;
        }
        return sum;
    }
}
```

Exercise #2

```
public class Exercise2 {
        //@ requires m>=0;
        //@ requires n>=0;
        //@ ensures \result ==m*n;
        int multiply ( int m, int n) {
                int product = 0;
                int i=0;
                //@ loop_invariant i>=0;
                //@ loop_invariant i<=n;
                //@ loop_invariant product==m*i;
                //@ decreasing n-i;
                while (i < n) {
                        product = add(product , m);
                        i=i+1;
                }
                return product ;
        }
}
```

Exercise #2

```
//@ ensures \result==left+right;
int add(int left, int right) {
        return left+right;
}
```

Exercise #3

```
public class Exercise3 {

    /*@ nullable @*/ Object object;

    //@ modifies this.object;
    void changeObject() {
        if (this.object==null) {
            this.object=null;
        } else {
            this.object=this.object;
        }
    }
}
```

Exercise #4

```
public int hashCode_1(Object o) {
    // @ asume o!=null;
    return o.hashCode();
}

public int hashCode_2(Object o) {
    // @ assert o!=null;
    return o.hashCode();
}
```

Exercise #5

```
class StringAppender {
      String str = "";

      //@ normal_behavior
      //@    requires o!=null;
      //@ also
      //@ exceptional_behavior
      //@    requires o==null;
      //@    signals_only ConcatException;
      public String append(/*@ nullable @*/Object o)
      throws ConcatException {
            if (o!=null) {
                  str += o. toString ();
            } else {
                  throw new ConcatException("Argument…");
            }
      }
}
```

Exercise #6

- Bag example we saw last class.

ESC/Java2: modular verifier

- Reasons over each method separately

```
Class MyArray {
  byte[] b;

  void createArray() { b = new byte[10]; }

  void storeArray() { createArray(); b[0]=1; }
}
```

- What is happening here?

ESC/Java2: modular verifier

- Reasons over each method separately

```
Class MyArray {
  byte[] b;
  //@ ensures b!=null && b.length==10;
  void createArray() { b = new byte[10]; }

  void storeArray() { createArray(); b[0]=1; }
}
```

- ESC/Java2 always deals with the callee contract, nor the implementation

ESC/Java2: incomplete

- ESC/Java2 may fail to prove all legal JML specifications

```
/*@ requires n>0;
  @ ensures \result==(\forall int n;
  @  (\exists int x,y,z ;
  @  Math.pow(x,n)+Math.pow(y,n)
  @  ==Math.pow(z,n)));
  @*/
Public boolean m(int n) { return true; }
```

- ESC/Java2 reports a warning

ESC/Java2: unsound

- ESC/Java2 sacrifices precision for performance

```
//@ invariant n>0;

public void increase()
   n++;
}
```

- ESC/Java2 reports no warning (should it?)

Example

```
class Purse {
  int money;
  //@ invariant money >= 0;
}
```

- Is method **loseMoney** correct?
- What will ESC/Java2 report?

```
class RichPerson {

  String mansion_address;
  Purse purse;

  //@ invariant purse.money > 10;

  public void earnMoney() {
    purse.money = purse.money +1;
  }
```

```
class PoorPerson {

  String slum_address;
  Purse purse;

  //@ invariant purse.money < 100;

  //@ requires purse.money > 0;
  public void loseMoney(RichPerson
my_rich_friend) {
    purse.money = purse.money -1;
  }
```

Counterexample

Possibly relevant items from the counterexample context:

typeof(brokenObj<4>) <: T_RichPerson

typeof(this) <: T_PoorPerson

(brokenObj<4>).(purse:7.25) == tmp0!purse:23.4

this.(purse:7.25) == tmp0!purse:23.4

(tmp0!purse:23.4).(money@pre:4.3.6) == 11

(tmp0!purse:23.4).(money:23.10) == 10

....

Counterexample heap view

Initial State

invariant purse.money > 10;

invariant purse.money < 100;

RichPerson

PoorPerso
n

this

Purse

Money =11

loseMoney()

Final State

invariant purse.money > 10;

invariant purse.money < 100;

RichPerson

PoorPerson

this

Purse

money=10

Example

```
class Purse {
  int money;
  //@ invariant money >= 0;
}
```

- Is method **loseMoney** correct?
- What will ESC/Java2 report?

```
class RichPerson {

  String mansion_address;
  Purse purse;

  //@ invariant purse.money > 10;

  public void earnMoney() {
    purse.money = purse.money +1;
  }
```

```
class PoorPerson {

  String slum_address;
  Purse purse;

  //@ invariant purse.money < 100;

  //@ requires purse.money > 0;
  public void loseMoney(RichPerson my_rich_friend) {
    purse.money = purse.money -1;
  }
```

Example

- ESC/Java2 con PoorPerson, RichPerson, Purse
  - PoorPerson: loseMoney() …
  - [0.197 s 39390680 bytes] passed
- ESC/Java2 found no bug:
- It did not verify if the invariant for RichPerson was preserved
  - ESC/Java2 believed it was not important

ESC/Java2: more limitations

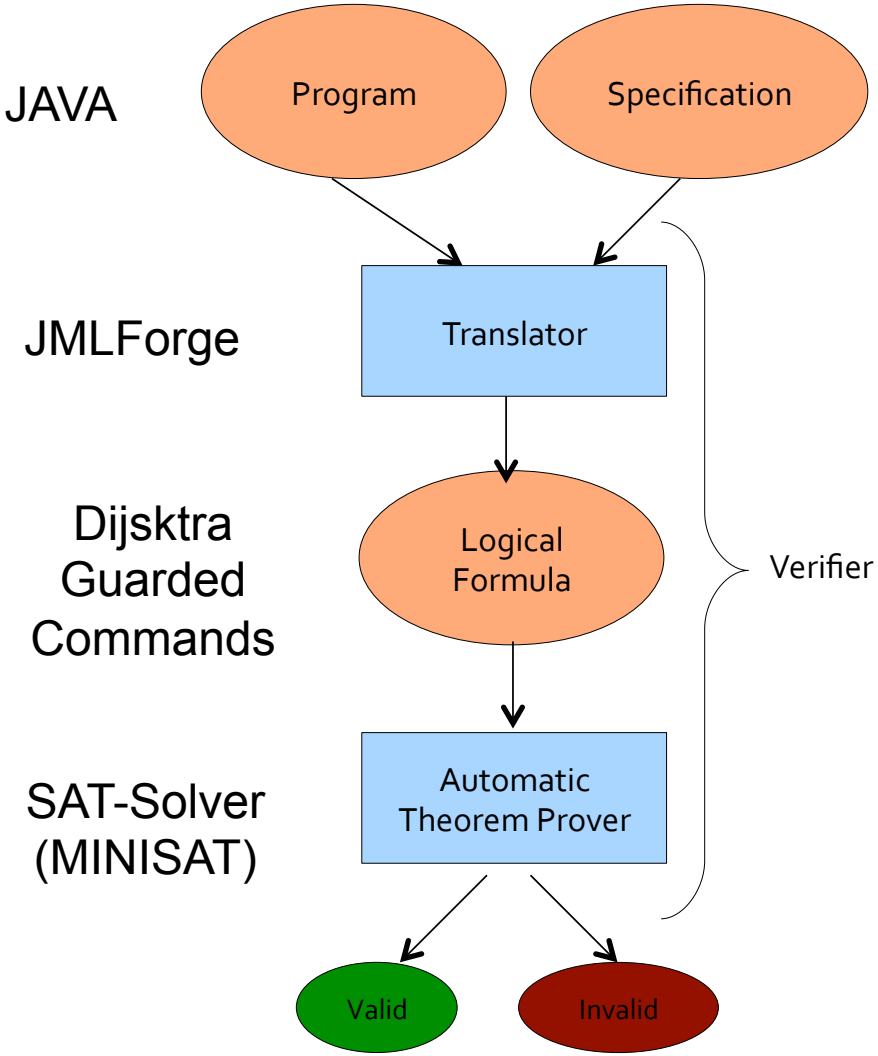
- Only some class invariants are checked at exiting a method
- \reach expressions are not supported.
- -LoopSafe for checking loop correctness, otherwise –Loop X for unrolling loops X times.

JMLForge

- Relies on a SAT-Solver instead of a SMT-Solver
- **Objective**: Find counterexamples within some fixed user-provided bound.
- http://sdg.csail.mit.edu/forge/jmlforge.html

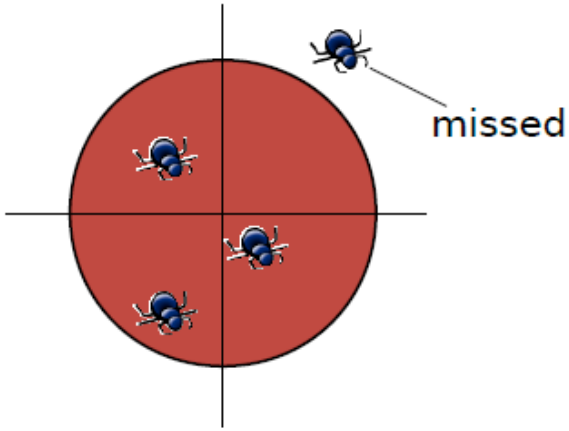
JMLForge

- Programming language
- Specification Language
- Logical representation of correctness
- Automatic decision procedure



JMLForge

- "Small-scope hypothesis"
  - Most bugs can be exhibited using small domains
- No validity proof
  - Search for a counterexample
- Traverses the representation of a **concrete search space**.
- We need to select a"scope" to check the programs



Bounded Verification

- We have to define a "scope" for each analysis
  - Maximum size of List domain
  - Maximum size of Node domain
  - Integer bitwidth
  - Maximum number of loop execution
    - Number of times it is allowed to execute a loop/recursion

- Because of this finitization:
  - Encode program and contract as SAT propositional formula => Feed to SAT-Solvers

JMLForge

- Modular as ESC/Java2
- No support for intraprocedural annotations
  - //@ assume
  - //@ set ghost fields
  - // @ loop_invariant

ESC/Java2 vs. JMLForge

ESC/Java 2	JML Forge
Proves	Searchs counterexample
No Failure => Contract holds (explicit or implicit) (*)	No Failure => No contract errors within scope
Failure => potential bug (or proof was inconclusive)	Failure => bug exists and counterexample is valid
Simple data structures (arrays, integer, plain objects) Arithmetics	Complex (but small) data structures
Quantifiers	Quantifiers: Reachability operator

(*) Consider sources of unsoundness

Some references

- ESC/Java2
  - http://kindsoftware.com/products/opensource/ESCJava2/
- JMLForge
  - http://sdg.csail.mit.edu/forge/jmlforge.html

Automated Testing & Verification

Modular verification of programs
(Part 1)

Juan Pablo Galeotti, Alessandra Gorla, Andreas Rau
Saarland University, Germany

Object Invariant semantics

- An object invariant is a property that holds on every <u>visible state</u> of an object.
- What is a visible state?
  - The pre and post state of an invocation to a method of that object

```
class Test
   public void Main()
  {
    Counter d = new Counter();
    d.Inc();
    d.Inc();
  }
}
```

inv(d)
Always holds

```
class Counter
  int c;
  invariant c>=0;
  public Counter() {
    c=0;
  }
  public void Inc()
    modifies c;
    ensures c=old(c)+1; {
    c++;
  }
}
```

Modularity

- When we verify a method C.M() :
  - **Assume** that **ALL** invariants of all pre existitng objects hold at the method entry.
  - **Prove** that **ALL** invariants of all existing objects at the method exit hold
- When we invoke method C′.M′() from method C.M():
  - **Prove** that ALL invariants of all pre-existing objects hold before executing the method.
  - **Assume** ALL invariants of all existing objects hold after executing the method.

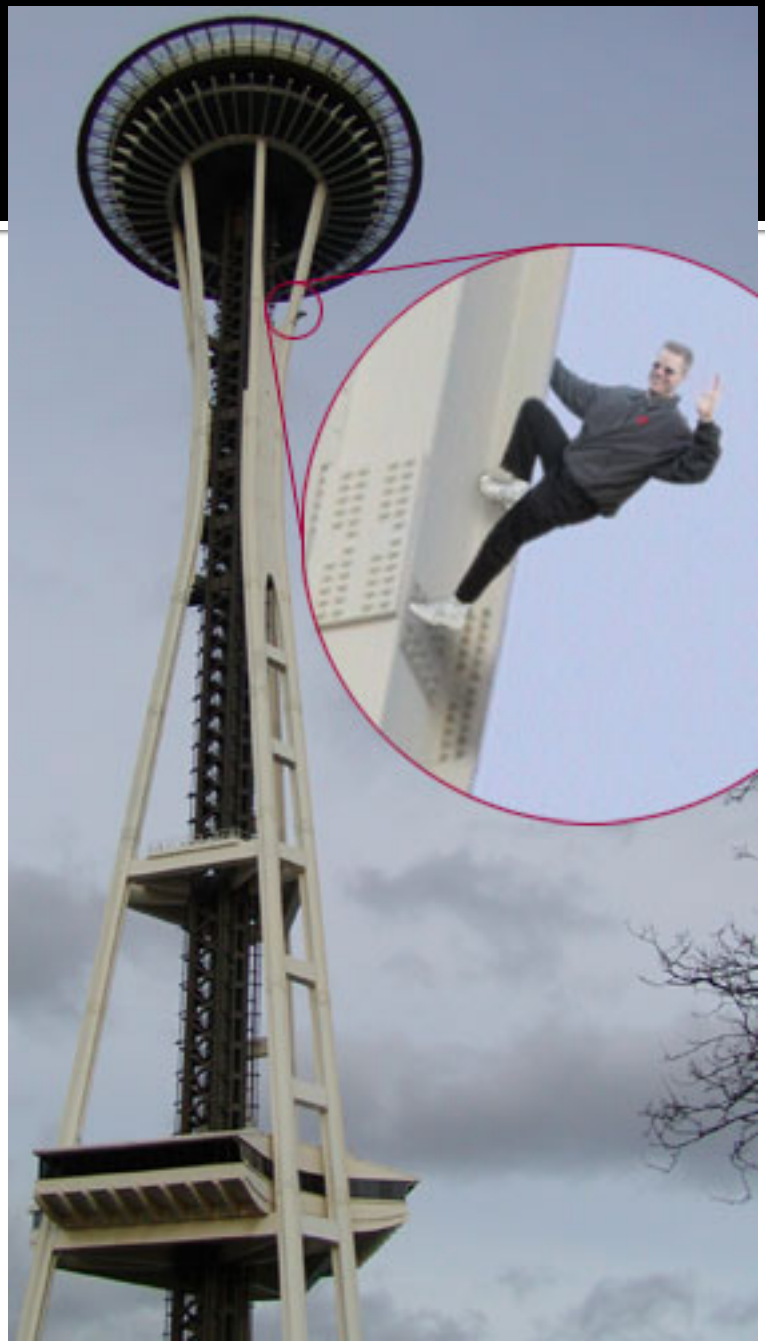
But this semantics is not modular

Modular Verification

- We would like to prove that all object invariants are preserved, but modularly:
  - We verify the object invariant of the class under analysis
  - We preserve by construction the invariant for other objects

Modular Verification

- Spec# proposes a methodology for verifying invariants modularly.



About Spec#

- Extension of C#
  - non-null types, preconditions, postconditions, object invariants
- Boogie intermediate verification language (also used in HAVOC, VCC)
- Decision procedure: Z3 SMT-Solver
- Have fun!
  - http://rise4fun.com/SpecSharp/

Object invariants (Recall)

```
class Counter {
  int c;
  bool even;
  invariant c>=0;
  invariant even <==> c % 2==0;
  public Counter() {
    c=0; even:=true;
  }
  public void Inc()
    modifies c;
    ensures c=old(c)+1; {
    c++;
    even=!even;
  }
}
```

inv established and verified

Does not hold inv

inv verified

Key message:
An object invariant can be invalid temporarily

Reentrancy

```
class Counter {
  …
  invariant c>=0;
  invariant even <==> c % 2==0;
  …

  public void Inc()
    modifies c;
    ensures c=old(c)+1; {
    c++; // inv does not hold
    m();
    even=!even; // inv checked
  }

  public m()
  {…}
```

What happens with m() ?

Modular verification of invariants in Spec#

Verifying object invariants modularly is not free:

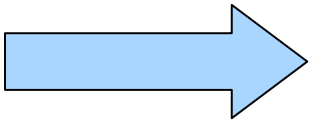
- We have restrictions on the invariants we might write
- An certain protocol for updating must be obeyed
- We have to take special care of data structures

Update protection

- A new ghost field named inv is introduced:
  - Managed by the translator/verifier (not by the programmer)
  - Possible values { Mutable, Valid }

- **Mutable**: invariant might not hold and fields can be updated
- **Valid**: object invariant must hold
  - ∀ o : o.inv=Mutable || invariant(o)

```
o.f := e ;
```



```
assert o.inv=Mutable;
o.f := e ;
```

- Objects can only be modified if we "expose" them

Expose blocks

```
class Counter
  int c;
  bool even;
  invariant c>=0;
  invariant even <==> c % 2==0;
  public Counter() {
    c=0; even:=true;
  }
  public void Inc()
    modifies c;
    ensures c=old(c)+1; {
    expose(this) {
      c++;
      even=!even;
    }
  }
```

Unpack:
1: assert inv==valid;
2: this.inv=mutable

Update is possible since the object is in mutable state

Pack:
1: assert inv==mutable;
2: assert invariant
3: this.inv=valid

Expose blocks & reentrancy

```
class Counter {
  invariant c>=0 && even <==> c % 2==0;
  public void Inc()
    // implicit requires this.inv=Valid
    modifies c;
    ensures c=old(c)+1; {
    expose(this) {
      c++;
      m();
      even=!even;
    }
  }
  public void m()
    // implicit requires this.inv=Valid
  { … }
}
```

Change this.inv from valid to mutable

Error: this.inv!=Valid

Pack / Unpack

The semantics of **`expose(o){Q}`**, is defined as:

```
unpack o;
Q
pack o;
```

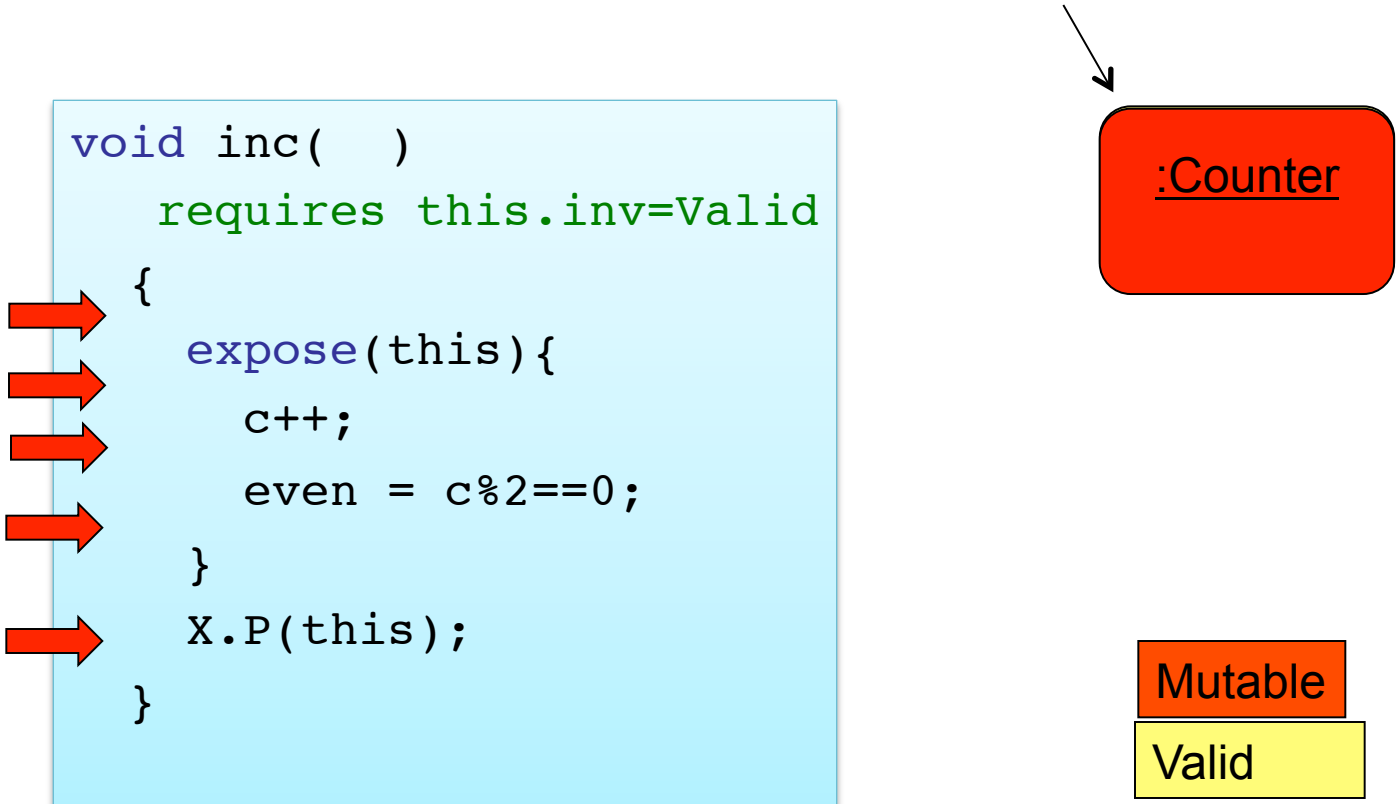
- *unpack*(o) turns object *o* into "mutable"
- *pack*(o) object invariant is reestablished and turns object *o* into ”valid”

unpack o :	pack o :

assert o.inv == Valid;
o.inv := Mutable
	assert o.inv = Mutable;
	assert Inv(o);
	o.inv := Valid

The expose {...} protocol

```
void inc(  )
   requires this.inv=Valid
  {
    expose(this){
      c++;
      even = c%2==0;
    }
    X.P(this);
  }
```



The expose semantics is defined ad :
expose(o) s; = unpack o; s; pack o;

Some restrictions:

- The Valid-Mutable protocol
  - Only object field accesses are allowed:
  - this.f1, this.f2, …
- Example:
  - invariant this.c>=0 && this.even <==> this.c%2==0 ;