Verifying the State Design Pattern using Object Propositions

Ligia Nistor

Computer Science Department

Carnegie Mellon University

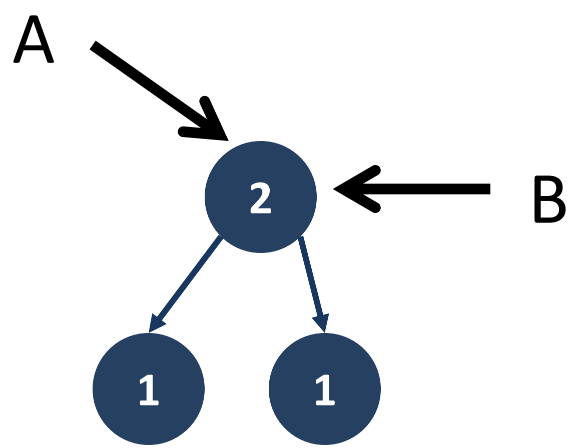
Why verify programs?

- **Verification** vs. debugging
- **Verification at compile time** vs. testing at run time

Formal verification

- Use formal rules to reason about correctness of programs
- Difficult because of aliasing

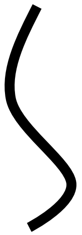
Reference A depends on property



Reference B can break property

Object Propositions

- New verification methodology 

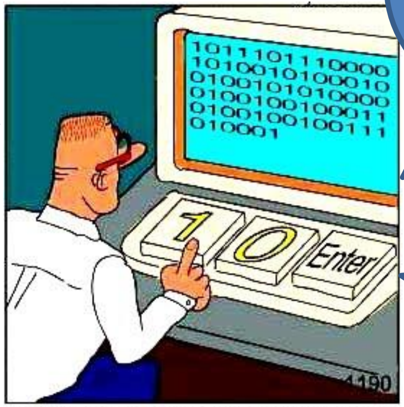
- Express specifications about objects ➔ object propositions
- Modularity ➔ verify classes independently
- Single-thread 

Object propositions

I want to verify that my program satisfies this property

Then I need to provide a specification

I'll write the specification using object propositions



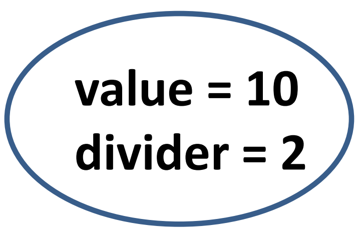
REAL Programmers code in BINARY.

Object proposition: abstract predicate + fractional permission

Abstract Predicates

- Predicate **MultipleOf(int a)** = the *divider* field of this object == a && the *value* field is a multiple of *divider*

obj

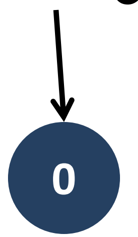


obj satisfies MultipleOf(2)

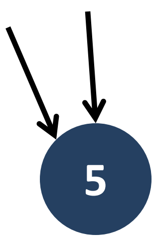
[M. Parkinson]

Fractional permissions

- dealing with aliases



permission of 1
read/write access



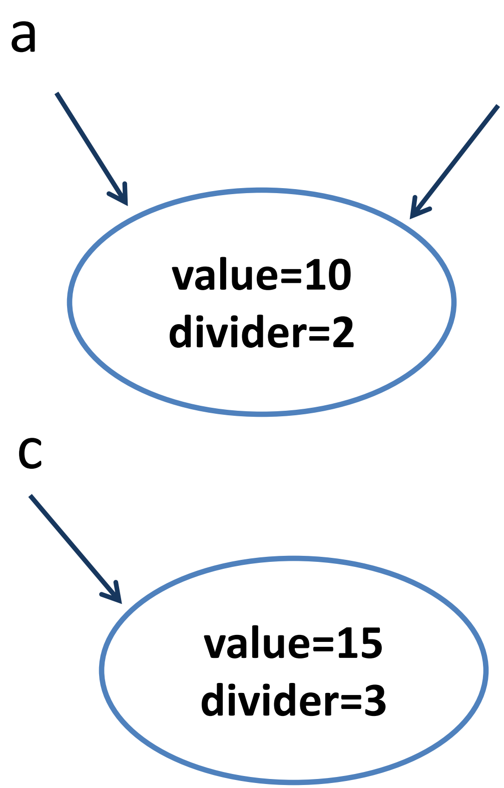
permission of 1/2
read/write access,
as long as the initial
predicate is maintained

Contribution: The state referred to by a fraction < 1 is not immutable.
That state satisfies an invariant that can be relied on by other objects.

[Boyland]

Putting it together

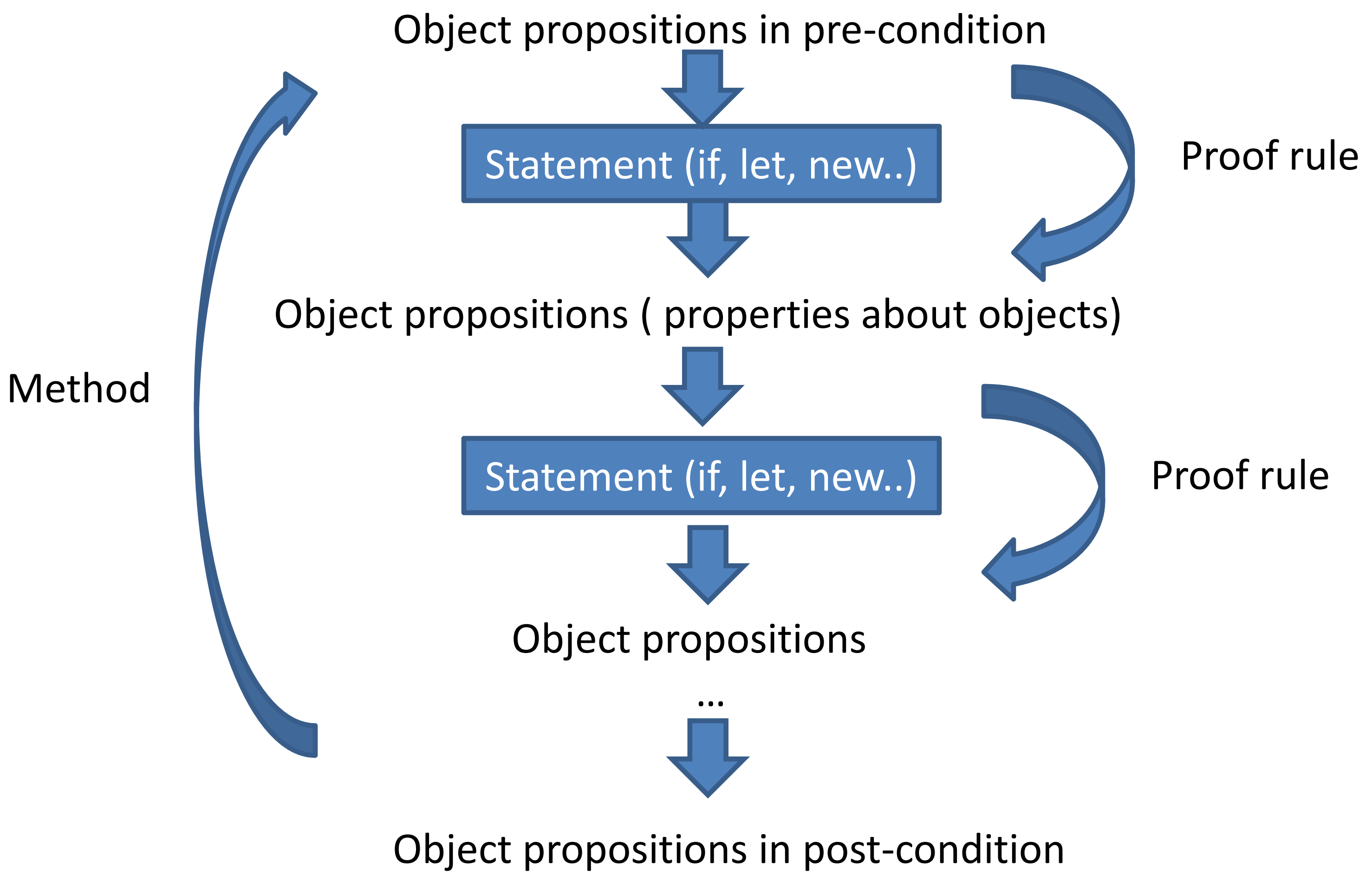
- Object proposition =
  abstract predicate + fractional permission



- a#1/2 MultipleOf(2)
- c#1 MultipleOf(3)

The Verification of a Method

- Using proof rules



Linear logic

- Classical logic: from A and (A ⇒ B) get (A ∧ B)

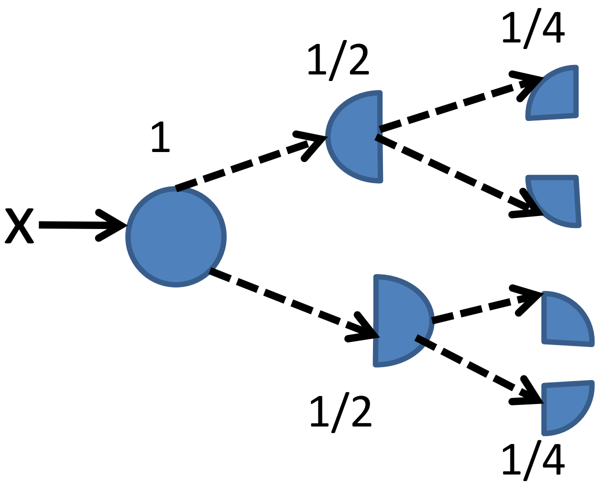
A

B

- Linear logic: from A and (A ⊸ B) get B (**transform**)
    - Logic of resources
    - ⊗ **Simultaneous** occurrence of resources
    - ⊕ **Alternative** occurrence of resources
- Object propositions = resources consumed upon usage

Formal system

- Rules for **splitting/adding** fractions



- x#1 ⇔ x#1/2 ⊗ x#1/2
- x#k ⇔ x#k/2 ⊗ x#k/2

[Boyland]

Pack, unpack

- Abstraction:

  Predicate: from outside ➔ MultipleOf(c)

  from inside ➔ get to the fields

- **pack** to a predicate

- **unpack** a predicate: gain access to fields of object

Consistency

- packed predicate → consistent
- unpacked predicate → inconsistent
- In a method, after the first assignment to a field, the unpacked predicate is inconsistent
- We have aliasing and fractions, how come unpacking is still sound?
- As long as we have a fraction to an object, we know that the invariant of that object will not be broken. When we pack back the predicate, the invariant is restored.
- We assume termination, so at end of program all objects are packed

Invariants

- Invariants are predicates that always hold at the boundary of methods, for all references pointing to the same object.
- Aliased objects can only depend on invariants, not on any kind of predicates.

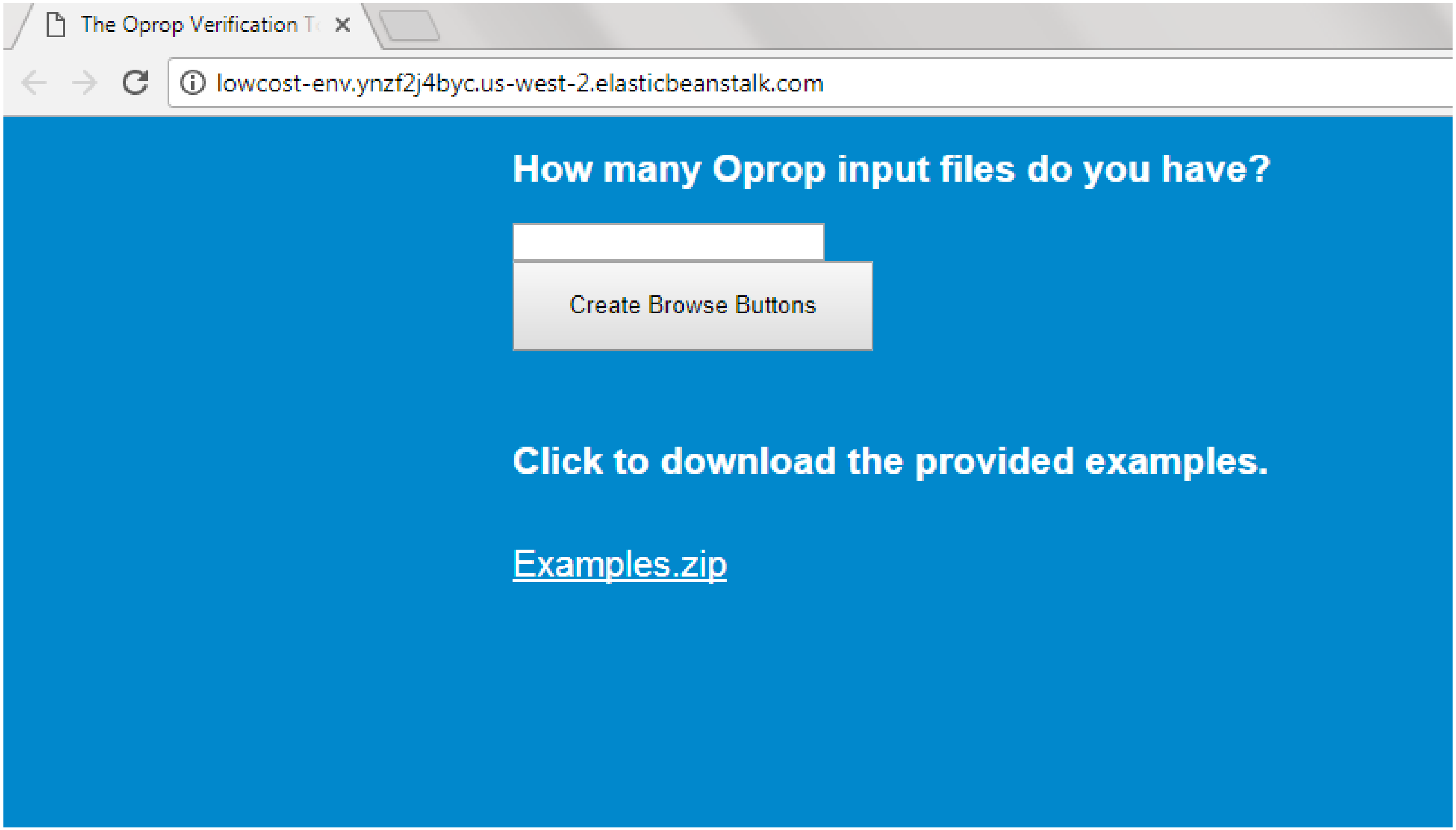
Oprop Grammar

$$
\begin{aligned}
\text{Prog} &::= \overline{\text{InterfDecl}}\ \overline{\text{ClDecl}}\ \text{e} \\
\text{InterfDecl} &::= \texttt{interface}\ I\ \{\ \overline{\text{InterfPredDecl}}\ \overline{\text{InterfMthDecl}}\ \} \\
\text{InterfPredDecl} &::= \texttt{predicate}\ Q(\overline{\text{T x}}) \\
\text{InterfMthDecl} &::= \text{T}\ m(\overline{\text{T x}})\ \text{MthSpec} \\
\text{ClDecl} &::= \texttt{class}\ C\ (\texttt{implements I})?\ \{\ \overline{\text{FldDecl}}\ \overline{\text{PredDecl}}\ \overline{\text{MthDecl}}\ \} \\
\text{FldDecl} &::= \text{T}\ f \\
\text{PredDecl} &::= \texttt{predicate}\ Q(\overline{\text{T x}}) \equiv \text{R} \\
\text{MthDecl} &::= \text{T}\ m(\overline{\text{T x}})\ \text{MthSpec}\ \{\ \overline{\text{e}};\ \texttt{return}\ \text{e}\ \} \\
\text{MthSpec} &::= \text{R} \multimap \text{R} \\
\text{R} &::= \text{P} \mid \text{R} \otimes \text{R} \mid \text{R} \oplus \text{R} \mid \\
&\quad \exists \text{x:T.R} \mid \exists \text{z:double.R} \mid \exists \text{z:double.z binop t} \Rightarrow \text{R} \mid \\
&\quad \forall \text{x:T.R} \mid \forall \text{z:double.R} \mid \forall \text{z:double.z binop t} \Rightarrow \text{R} \mid \\
&\quad \text{t binop t} \Rightarrow \text{R}
\end{aligned}
$$

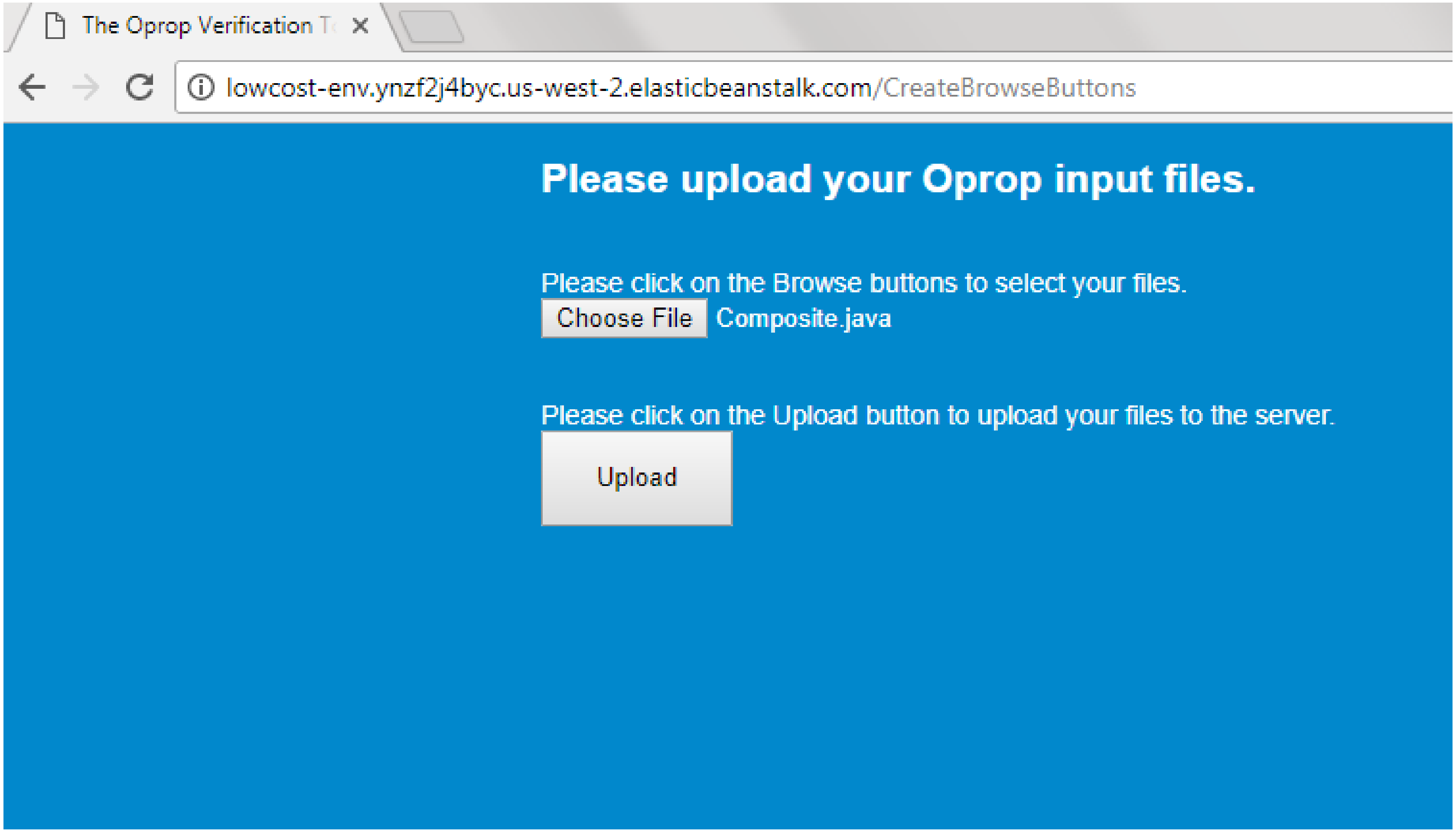
Oprop Grammar – cont.

$$
\begin{array}{rcl}
\mathsf{P} & ::= & r@\mathsf{k}\ Q\,(\bar{\mathsf{t}}) \mid \mathtt{unpacked}\,(r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})) \mid \\
 & & r.f \rightarrow \mathsf{x} \mid \mathsf{t}\ \mathsf{binop}\ \mathsf{t} \\
\mathsf{k} & ::= & \frac{n_1}{n_2}\ (\text{where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \mid \mathsf{z} \\
\mathsf{e} & ::= & \mathsf{t} \mid r.f \mid r.f = \mathsf{t} \mid r.m\,(\bar{\mathsf{t}}) \mid \\
 & & \mathtt{new}\ C\,(Q\,(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}])\ (\bar{\mathsf{t}}) \mid \\
 & & \mathtt{if}\ (\mathsf{t})\ \{\ \mathsf{e}\ \}\ \mathtt{else}\ \{\ \mathsf{e}\ \} \mid \mathtt{let}\ \mathsf{x} = \mathsf{e}\ \mathtt{in}\ \mathsf{e} \mid \\
 & & \mathsf{t}\ \mathsf{binop}\ \mathsf{t} \mid \mathsf{t}\ \&\&\ \mathsf{t} \mid \mathsf{t}\ \|\ \mathsf{t} \mid\ !\ \mathsf{t} \mid \\
 & & \mathtt{pack}\ r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}]\ \mathtt{in}\ \mathsf{e} \mid \mathtt{unpack}\ r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}]\ \mathtt{in}\ \mathsf{e} \\
\mathsf{t} & ::= & \mathsf{x} \mid n \mid \mathtt{null} \mid \mathtt{true} \mid \mathtt{false} \\
\mathsf{x} & ::= & r \mid i \\
\mathsf{binop} & ::= & + \mid - \mid \% \mid = \mid\ != \mid \leq \mid < \mid \geq \mid > \\
\mathsf{T} & ::= & C \mid \mathtt{int} \mid \mathtt{boolean} \mid \mathtt{double}
\end{array}
$$

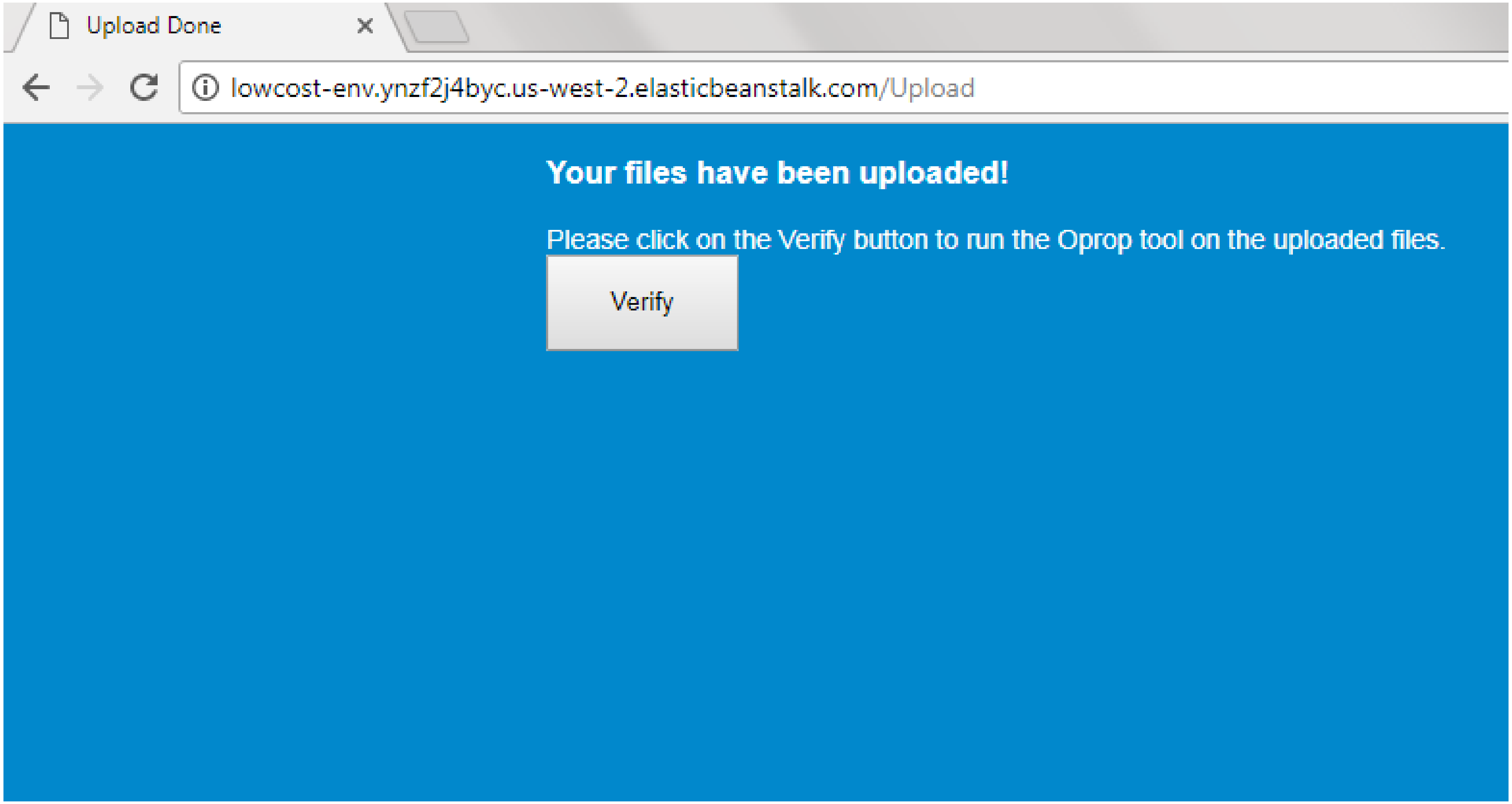
Oprop Online Tool – 1^{st} webpage



Oprop Online Tool – 2nd webpage



Oprop Online Tool – 3rd webpage



Oprop Online Tool – 4^{th} webpage

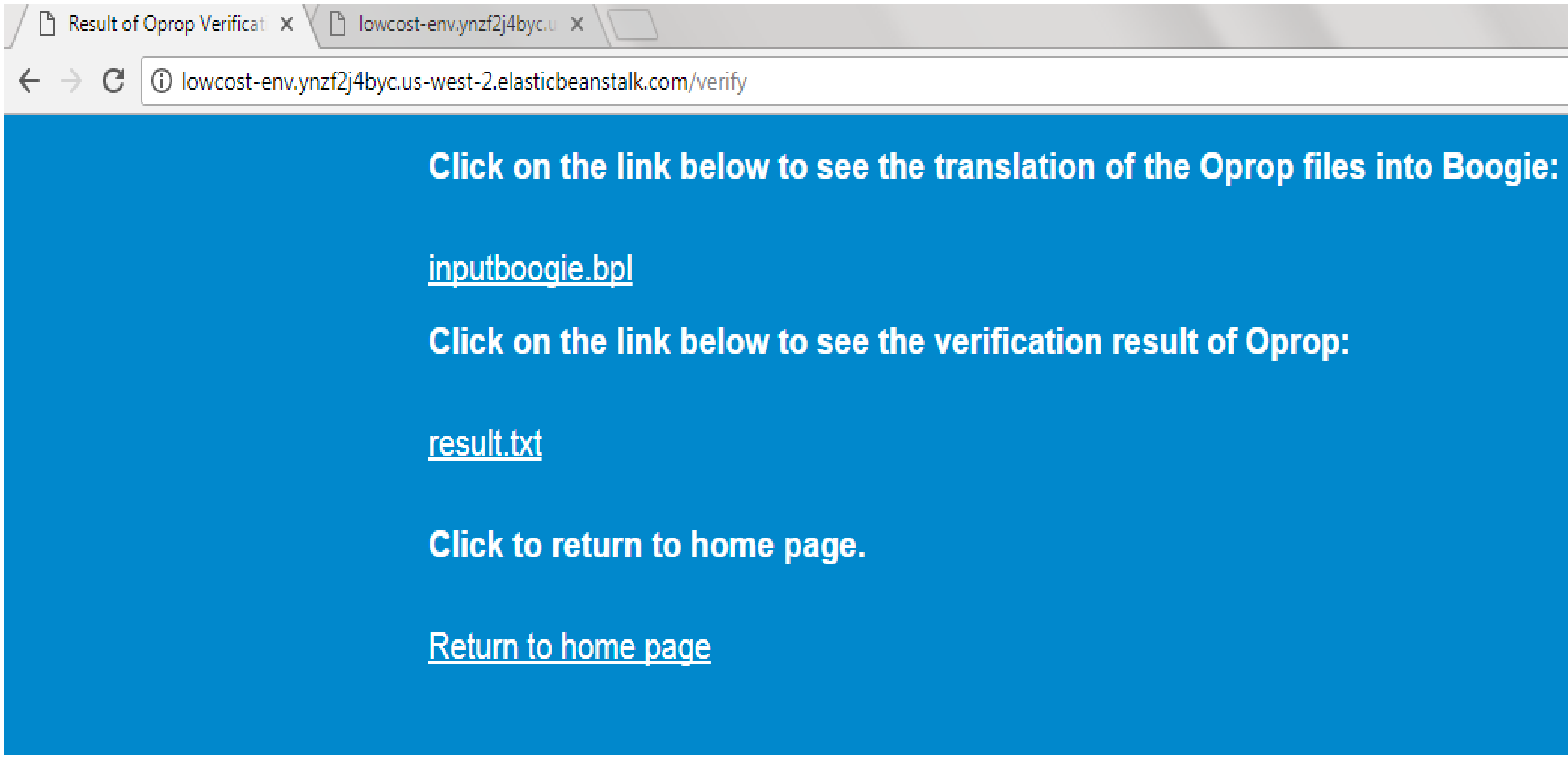
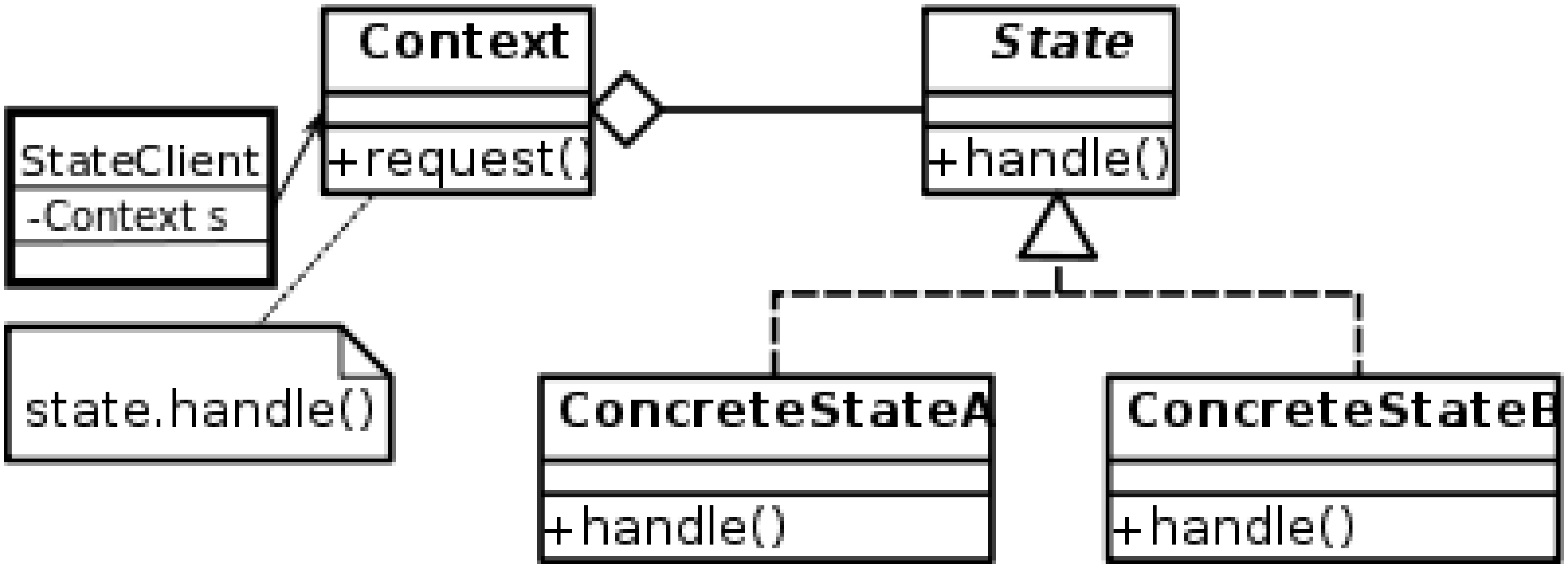
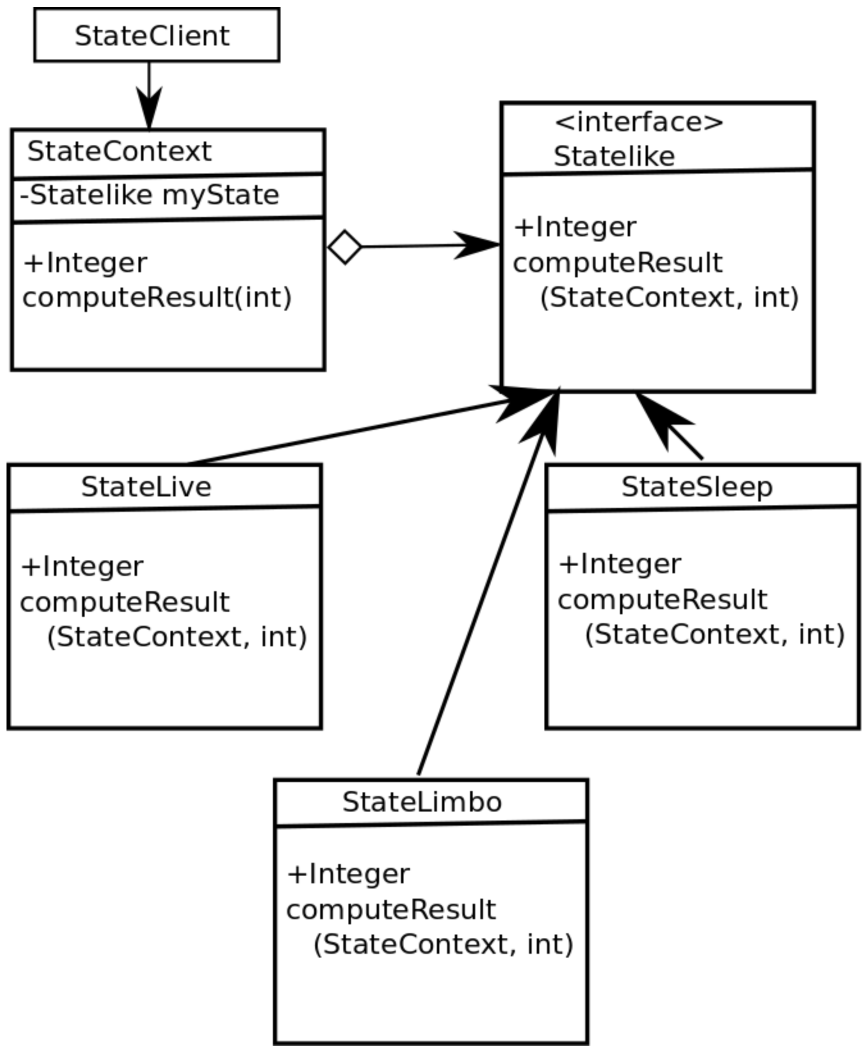


Diagram of State Pattern



My Example of the State Pattern



Class IntCell

```
class IntCell {
  int divider;
  int value;

  predicate BasicIntCell()=exists int divi,
    int val : this.divider -> divi &&
    this.value -> val

  predicate MultipleOf(int a)=exists int v:
    this.divider -> a && this.value -> v
   && ( (v - int(v/a)*a ) == 0 )

  IntCell(int divider1, int value1)
    ensures this.value == value1;
    ensures this.divider == divider1;
  {
    this.value  = value1;
    this.divider = divider1;
  }

```

Interface Statelike

```
interface Statelike {
  predicate StateMultipleOf3();

  IntCell computeResult(
    StateContext context, int num);
  ~double k, k2:
    requires (context#k stateContextMultiple3())
    ensures (context#k stateContextMultiple3())

  boolean checkMod3();
  ~double k:
    requires this#k StateMultipleOf3()
    ensures this#k StateMultipleOf3()
}
```

Class StateLive

```
class StateLive implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() =
    exists IntCell c, double k :
    this.cell -> c && (c#k MultipleOf(21))

  StateLive()
  {
    IntCell temp = new IntCell(0);
    this.cell = new StateLive(temp);
  }

  StateLive(IntCell c)
  ensures this.cell == c;
  { this.cell = c; }
```

Class StateLive – cont.

```
Statelike computeResult(
    StateContext context, int num)
~double k, k2:
    requires (context#k StateContextMultiple3())
    ensures (context#k StateContextMultiple3()) &&
        (context#k2 StateLimbo())
{
    IntCell i1 = new
        IntCell(MultipleOf(33)[num*33])(33, num*33);
    StateLike r = new
        StateLimbo(StateMultipleOf3()[i1])(i1);
    context.setState3(r);
    return r;
}

boolean checkMod3()
~double k:
    requires this#k StateMultipleOf3()
    ensures this#k StateMultipleOf3()
{
unpack(this#k StateMultipleOf3());
boolean temp =
  (this.cell.getValueInt() % 3 == 0);
pack(this#k StateMultipleOf3());
return temp;
}
```

Classes StateLimbo and StateSleep

```
class StateSleep implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(15))
```

```
class StateLimbo implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(33))
```

Class StateContext

```
class StateContext {
  Statelike myState;

  predicate StateContextMultiple3() =
    exists StateLike m, double k :
    this.myState -> m && (m#k StateMultipleOf3())

  StateContext(Statelike newState)
  ensures this.myState == newState;
  {
    this.myState = newState;
  }

  void setState3(Statelike newState)
  ~double k1, k2:
   requires this#k1 StateContextMultiple3()
   requires newState#k2 StateMultipleOf3()
   ensures this#k1
     StateContextMultiple3()[newState]
  {
   unpack(this#k1 StateContextMultiple3());
   this.myState = newState;
   pack(this#k1
     StateContextMultiple3())[newState];
  }
```

Class StateContext – cont.

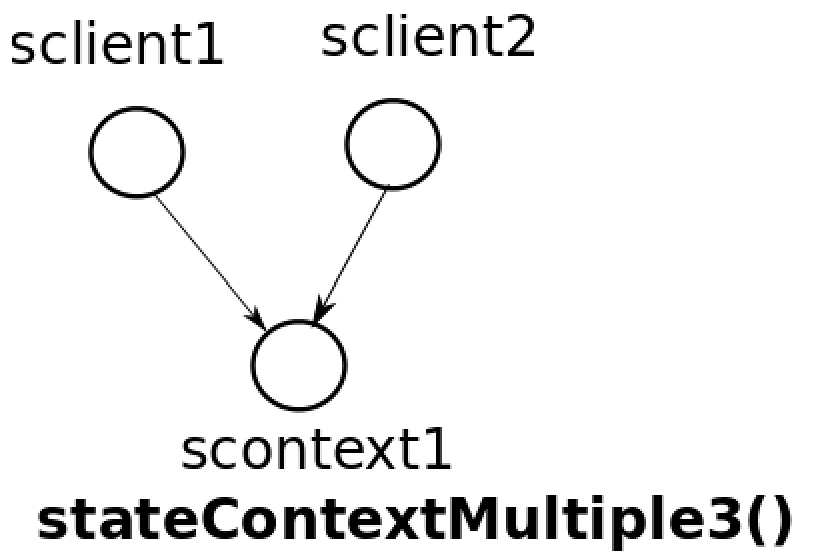
```
IntCell computeResultSC(int num)
~double k1, k2:
  requires (this#k1 StateContextMultiple3())
  ensures (this#k1 StateContextMultiple3())
{
  unpack(this#k1 stateClientMultiple3());
  IntCell temp =
    this.myState.computeResult(this, num);
  pack(this#k1 stateClientMultiple3());
  return temp;
}

boolean stateContextCheckMultiplicity3()
~double k:
  requires this#k StateContextMultiple3()
  ensures this#k StateContextMultiple3()
{
  unpack(this#k StateContextMultiple3())
  boolean temp=this.myState.checkMod3();
  pack(this#k StateContextMultiple3())
  return temp;
}
```

Class StateClient

```
void main()
~double k:
{
 IntCell i1 = new
   IntCell(MultipleOf(21))(21);
 Statelike st1 =
   new StateLive(StateMultipleOf3())(i1);
 StateContext scontext1 =
   new StateContext(
       stateContextMultiple3()[])(st1);
 StateClient sclient1 =
   new StateClient(
      stateClientMultiple3()[])(scontext1);
 StateClient sclient2 =
   new StateClient(
      stateClientMultiple3()[])(scontext1);
 scontext1.computeResultSC(1);
 sclient1.stateClientCheckMultiplicity3();
 scontext1.computeResultSC(2);
 sclient2.stateClientCheckMultiplicity3();
 scontext1.computeResultSC(3);
 sclient1.stateClientCheckMultiplicity3();
}
```

main() function in StateClient class



Implementation and code on GitHub

- https://github.com/ligianistor/boogie/blob/master/statelatest.bpl
- https://github.com/ligianistor/Oprop

Related work

- Bierhoff and Aldrich: access permissions
- Boyland: fractional permissions
- Parkinson: abstract predicates

- Barnett & Leino: Boogie verifier
- Krishnaswami: higher-order separation logic
- Nanevski: Hoare Type Theory
- Jacobs, Leino, Smans: multi-threaded OO programs

Future Work

- Augment features of Oprop language so that state pattern can be verified using Oprop
- Extend for multi-threaded programs

Conclusions

- **Object proposition** = abstract predicate + fractional permission
- Verified instance of **State Design Pattern**