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

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

\[
\text{obj}
\]

\[
\text{value} = 10 \\
\text{divider} = 2
\]

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
  \[
  \text{value=10} \\
  \text{divider=2}
  \]
  \[
  \text{value=15} \\
  \text{divider=3}
  \]

  c

  • \text{a#1/2 \text{MultipleOf}(2)}
  • \text{c#1 \text{MultipleOf}(3)}
The Verification of a Method

• Using proof rules

Object propositions (properties about objects)

Statement (if, let, new..)

Object propositions in pre-condition

Proof rule

Method

Object propositions (properties about objects)

Statement (if, let, new..)

Object propositions

Proof rule

Object propositions in post-condition
Linear logic

• Classical logic: from A and \((A \Rightarrow B)\) get \((A \land B)\)

• Linear logic: from A and \((A \rightarrow B)\) get B \hspace{1cm} \text{(transform)}

• Logic of resources
  • \(\otimes\) **Simultaneous** occurrence of resources
  • \(\oplus\) **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 $\rightarrow$ consistent
- unpacked predicate $\rightarrow$ 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

Prog ::= InterfDecl CliDecl e

InterfDecl ::= interface I { InterfPredDecl InterfMthDecl }

InterfPredDecl ::= predicate Q(Tx)

InterfMthDecl ::= T m(Tx) MthSpec

CliDecl ::= class C (implements I)? { FldDecl PredDecl MthDecl }

FldDecl ::= T f

PredDecl ::= predicate Q(Tx) ⇔ R

MthDecl ::= T m(Tx) MthSpec { e; return e }

MthSpec ::= R → R

R ::= P | R ⊗ R | R ⊕ R |
    | 3x:T.R | 3z:double.R | 3z:double.z binop t ⇒ R |
    | ∀x:T.R | ∀z:double.R | ∀z:double.z binop t ⇒ R |
    | t binop t ⇒ R
Oprop Grammar – cont.

\[ P ::= r @ k \tilde{Q}(\tilde{t}) \mid \text{unpacked}(r @ k \tilde{Q}(\tilde{t})) \mid r.f \rightarrow x \mid t \text{ binop } t \]

\[ k ::= \frac{n_1}{n_2} \text{ (where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \mid z \]

\[ e ::= t \mid r.f \mid r.f = t \mid r.m(\tilde{t}) \mid \]
\[ \text{new } C(Q(\tilde{t})[\tilde{t}])(\tilde{t}) \mid \]
\[ \text{if } (t) \{ e \} \text{ else } \{ e \} \mid \text{let } x = e \text{ in } e \mid \]
\[ t \text{ binop } t \mid t \& \& t \mid t || t \mid ! t \mid \]
\[ \text{pack } r @ k \tilde{Q}(\tilde{t})[\tilde{t}] \text{ in } e \mid \text{unpack } r @ k \tilde{Q}(\tilde{t})[\tilde{t}] \text{ in } e \]

\[ t ::= x \mid n \mid \text{null} \mid \text{true} \mid \text{false} \]

\[ x ::= r \mid i \]

\[ \text{binop} ::= + \mid - \mid \% \mid = \mid != \mid \leq \mid < \mid \geq \mid > \]

\[ T ::= C \mid \text{int} \mid \text{boolean} \mid \text{double} \]
Oprop Online Tool – 1st webpage

How many Oprop input files do you have?

Create Browse Buttons

Click to download the provided examples.

Examples.zip
Oprop Online Tool – 2nd webpage

Please upload your Oprop input files.

Please click on the Browse buttons to select your files.
Choose File: Composite.java

Please click on the Upload button to upload your files to the server.
Upload
Oprop Online Tool – 3rd webpage

Your files have been uploaded!

Please click on the Verify button to run the Oprop tool on the uploaded files.

Verify
Oprop Online Tool – 4th webpage

Click on the link below to see the translation of the Oprop files into Boogie:

inputboogie.bpl

Click on the link below to see the verification result of Oprop:

result.txt

Click to return to home page.

Return to home page
Diagram of State Pattern
My Example of the State Pattern

[Diagram showing the State Pattern with classes and methods]

- StateClient
- StateContext
  - Statelike myState
  - Integer computeResult(int)
- <interface> Statelike
  + Integer computeResult (StateContext, int)
- StateLive
  + Integer computeResult (StateContext, int)
- StateSleep
  + Integer computeResult (StateContext, int)
- StateLimbo
  + Integer computeResult (StateContext, int)
Class IntCell

```java
1 class IntCell {
2     int divider;
3     int value;
4
5     // predicate
6     predicate BasicIntCell()=exists int divi,
7         int val : this.divider -> divi &&
8         this.value -> val
9
10    // predicate
11    predicate MultipleOf(int a)=exists int v:
12         this.divider -> a && this.value -> v
13         && ( (v - int(v/a)*a ) == 0 )
14
15    IntCell(int divider1, int value1)
16         ensures this.value == value1;
17         ensures this.divider == divider1;
18     {
19         this.value = value1;
20         this.divider = divider1;
21     }
22}
```
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.

18   Statelike computeResult(
19       StateContext context, int num)
20     ~double k, k2:
21         requires (context#k StateContextMultiple3())
22         ensures (context#k StateContextMultiple3()) &&
23             (context#k2 StateLimbo())
24     {
25         IntCell i1 = new
26             IntCell(MultipleOf(33)[num*33])(33, num*33);
27         StateLike r = new
28             StateLimbo(StateMultipleOf3()[i1])(i1);
29         context.setState3(r);
30         return r;
31     }
32
33     boolean checkMod3()
34     ~double k:
35         requires this#k StateMultipleOf3()
36         ensures this#k StateMultipleOf3()
37     {
38         unpack(this#k StateMultipleOf3());
39         boolean temp =
40             (this.cell.getValueInt() % 3 == 0);
41         pack(this#k StateMultipleOf3());
42         return temp;
43     }
Classes StateLimbo and StateSleep

```
1  class StateSleep implements Statelike {
2      IntCell cell;
3
4      predicate StateMultipleOf3() = exists IntCell c, double k :  
5          this.cell -> c && (c#k MultipleOf(15))
```

```
1  class StateLimbo implements Statelike {
2      IntCell cell;
3
4      predicate StateMultipleOf3() = exists IntCell c, double k :  
5          this.cell -> c && (c#k MultipleOf(33))
```
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.

```plaintext
1  IntCell computeResultSC(int num)
2  ~double k1, k2:
3  \textit{requires} (this\#k1 StateContextMultiple3())
4  \textit{ensures} (this\#k1 StateContextMultiple3())
5  {
6      unpack(this\#k1 stateClientMultiple3());
7      IntCell temp =
8      this.myState.computeResult(this, num);
9      pack(this\#k1 stateClientMultiple3());
10     return temp;
11  }
12
13  boolean stateContextCheckMultiplicity3()  
14  ~double k:
15  \textit{requires} this\#k StateContextMultiple3()  
16  \textit{ensures} this\#k StateContextMultiple3()  
17  {
18      unpack(this\#k StateContextMultiple3())
19      boolean temp = this.myState.checkMod3();
20      pack(this\#k StateContextMultiple3())
21     return temp;
22  }
```
Class StateClient

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

```
sclient1 --> scontext1
sclient2 --> scontext1

stateContextMultiple3()
```
Implementation and code on GitHub

- [https://github.com/ligianistor/boogie/blob/master/statelatest.bpl](https://github.com/ligianistor/boogie/blob/master/statelatest.bpl)
- [https://github.com/ligianistor/Oprop](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**