

# Data Abstraction in Spec# and Boogie

**Peter Müller**  
ETH Zürich  
Switzerland

Joint work with *Ádám Darvas* and *Rustan Leino*

2

## Spec#

- Experimental mix of contracts and tools
- Superset of C#
  - Non-null types
  - Pre- and postconditions
  - Object invariants
- Tool support
  - More type checking
  - Compiler-emitted run-time checks
  - Static program verification (Boogie)
- Here: sequential programs

## Data Abstraction using Methods

Needed for

- Subtyping
- Information hiding

```
interface Shape {
  pure int Width( );
  void DoubleWidth( )
  ensures Width( ) == old( Width( ) ) * 2;
}
```

```
class Rectangle implements Shape {
  int x1; y1; x2; y2;
  pure int Width( )
  private ensures result == x2 - x1; { ... }
  void DoubleWidth( )
  ensures Width( ) == old( Width( ) ) * 2;
  { ... } }
```

## Encoding of Pure Methods

- Pure methods are encoded as functions

$$M: \text{Value} \times \text{Value} \times \text{Heap} \rightarrow \text{Value}$$

- Functions are axiomatized based on specifications

$$\forall t, p, H: \text{Pre}_M(t, p, H) \Rightarrow \text{Post}_M(t, p, H) [ M(t, p, H) / \text{result} ]$$

## Problem 1: Inconsistent Specifications

- Flawed specifications potentially lead to **inconsistent axioms**
- Not always detected during verification
  - Abstract methods
  - Non-terminating methods
  - Unsoundness

```
class Inconsistent {
  pure int Wrong( )
  ensures result == 1;
  ensures result == 0;
  { ... }
}
```

```
class List {
  List next;
  pure int Len( )
  ensures result == Len( ) + 1;
  { ... }
  ... }
}
```

## Problem 2: Weak Purity

- Weak purity can be observed through reference equality
- Test for reference equality cannot be prevented easily
  - Needed for existing objects
  - Might happen in another method

```
class C {
  pure C Alloc( )
  ensures fresh( result );
  { return new C( ); }

  void Foo( )
  ensures Alloc( ) == Alloc( );
  { ... }
}
```

$Alloc( this, \mathcal{H} ) = Alloc( this, \mathcal{H} )$

## Problem 3: Frame Properties

- Result of pure methods depends on the heap

*Has( list, o,  $\mathcal{H}$  )*

- Different invocations typically refer to **different heaps**

```
class List {
  pure bool Has( object o ) { ... }
  void Remove( object o )
    requires Has( o );
  { ... }
  ... }

```

```
void Foo( List list, object o )
  requires list.Has( o );
{
  log.Log( "Message" );
  list.Remove( o );
}

```

## Data Abstraction using Model Fields

- Specification-only fields
- Value is determined by a mapping from concrete state
- Very similar to parameterless pure methods

```
interface Shape {
  model int width;
  void DoubleWidth( )
    ensures width == old( width ) * 2;
}

```

```
class Rectangle implements Shape {
  int x1; y1; x2; y2;
  model int width | width == x2 - x1;
  void DoubleWidth( )
    ensures width == old( width ) * 2;
  { ... } }

```

## Variant of Problem 3: Frame Properties

```
class Legend {
  Rectangle box; int font;
  model int mc | mc==box.width / font;
  ... }
```

```
class Rectangle {
  model int width | width == x2 - x1;
  void DoubleWidth( )
    modifies x2, width;
    ensures width = old( width ) * 2;
  { x2 := (x2 - x1) * 2 + x1; }
  ... }
```

- Assignment might change model fields of client objects
- Analogous problem for subtypes

## Background on Boogie Methodology

- Objects can be **mutable** or **valid**
  - Special field  $inv \in \{ mutable, valid \}$
- **Mutable**
  - Object invariant may not hold
  - Field updates allowed
- **Valid**
  - Object invariant holds
  - Field updates not allowed (proof obligation)
- $inv$  is changed by **special commands pack** and **unpack**

## Validity Principle

```
class List {
  List next;
  invariant list is acyclic;
  model int len | len == (next == null) ? 1 : next.len + 1;
  ... }
```

- Only model fields of **valid objects** have to satisfy their constraints

$$\forall X, m: X.\text{inv} = \text{valid} \Rightarrow R_m(X, X.m)$$

- Avoids inconsistencies** due to invalid objects

## Decoupling Principle

- Decoupling: Model fields are **not updated instantly** when dependee fields are modified
  - Values of model fields are **stored in the heap**
  - Updated when** object is being **packed**

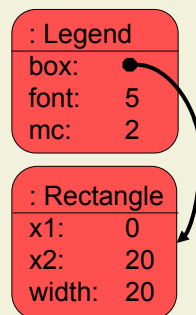
```
class Rectangle {
  model int width | width == x2 - x1;
  void DoubleWidth( ) requires inv==valid; {
    unpack this;
    x2 := (x2 - x1) * 2 + x1;
    pack this;
  }
  ... }
```

: Legend  
box: ●  
font: 5  
mc: 2

: Rectangle  
x1: 0  
x2: 20  
width: 20

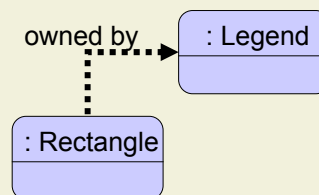
## Mutable Dependent Principle

- **Mutable Dependent:** If a model field  $o.m$  depends on a field  $x.f$ , then  **$o$  must be mutable whenever  $x$  is mutable**



## Mutable Dependents through Ownership

- Establish hierarchy (**ownership**) on objects
- **Ownership rule:** When an object is mutable, so are its (transitive) owners
- **Ownership requirement:** A constraint for a model field of object  $X$  may only depend on
  - The concrete fields of  $X$  and
  - The fields of objects (transitively) owned by  $X$

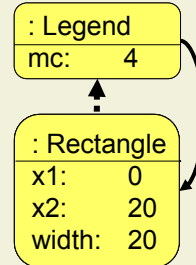


## The Methodology in Action

```

class Rectangle {
  void DoubleWidth( )
    requires inv == valid &&
             owner.inv == mutable;
    modifies width, x2;
  {
    unpack this;
    x2 := (x2 - x1) * 2 + x1;
    pack this;
  }
  ... }

```



```

class Legend {
  rep Rectangle box;
  model int mc |
    mc == box.width / font;
  ... }

```

## Automatic Updates of Model Fields

```

pack X ≡
  assert X ≠ null ∧ X.inv = mutable;
  assert Inv( X );
  assert ∀p: p.owner = X ⇒ p.inv = valid; ...
  X.inv := valid;
  foreach m of X:
    assert ∃r: Rm( X, r );
    X.m := choose r such that Rm( X, r );
  end

```

## Soundness

- Theorem:

$$\forall X, m: X.\text{inv} = \text{valid} \Rightarrow R_m( X, X.m )$$

- Proof sketch

- Object creation: new object is initially mutable
- Field update  $X.f := E$ ;  
Model fields of  $X$ : asserts  $X.\text{inv} = \text{mutable}$   
Model fields of  $X$ 's owners: mutable dependent principle
- Unpack  $X$ : changes  $X.\text{inv}$  to mutable
- Pack  $X$ : updates model fields of  $X$

## Problems Revisited

- Inconsistent specifications

```
model int wrong | wrong==0 && wrong==1;
```

```
model int len | len == next.len + 1;
```

Witness  
requirement

Ownership  
requirement

- Frame properties

```
modifies width, x2;
```

Validity,  
decoupling, and  
mutable dependent  
principle

- Weak purity

```
ensures alloc == alloc;
```

Decoupling  
principle

## Problem 1: Inconsistent Specifications

- Witness requirement for non-recursive specifications
- Ownership for traversal of object structures
- Termination measures for recursive specs

```
pure int Wrong( )
  ensures result == 1;
  ensures result == 0;
```

```
pure int Len( )
  ensures result == Len( ) + 1;
  measured_by height( this );
```

```
pure static int Fac( int n )
  requires n >= 0;
  ensures result ==
    ( n==0 ) ? 1 : Fac( n-1 ) * n;
  measured_by n;
```

## Problem 2: Modeling Weak Purity

- Explicit modeling of heap changes

$$M_H: \text{Value} \times \text{Value} \times \text{Heap} \rightarrow \text{Heap}$$

- Alloc( ) == Alloc( ) example revisited

$$\text{Alloc}( \text{this}, \mathcal{H} ) = \text{Alloc}( \text{this}, \text{Alloc}_H( \text{this}, \mathcal{H} ) )$$

- Axioms for postcondition and purity

$$\forall t, p, H: \text{Pre}_M( t, p, H ) \Rightarrow \text{Post}_M( t, p, M_H( t, p, H ) ) [ M( t, p, H ) / \text{result} ]$$

## Problems of Explicit Heap Functions

- Formulas become unwieldy
- Commutativity of operators is lost

**ensures**  $M( ) = N( )$ ;

$M( \text{this}, \mathcal{H} ) = N( \text{this}, M_H( \text{this}, \mathcal{H} ) )$

**ensures**  $N( ) = M( )$ ;

$N( \text{this}, \mathcal{H} ) = M( \text{this}, N_H( \text{this}, \mathcal{H} ) )$

- Matching specifications is difficult

**ensures**  $Q( ) \wedge R( )$ ;

$Q( \text{this}, \mathcal{H} ) \wedge R( \text{this}, Q_H( \text{this}, \mathcal{H} ) )$

**assert**  $R( )$ ;

$R( \text{this}, \mathcal{H} )$

## Problem 2: Restricted Weak Purity

- Pure methods must not return references to new objects
 

<pre>pure C Alloc( ) { return new C( ); }</pre>
---
- Provide value types for sets, sequences, etc.
- Case study
  - 8 out of 182 pure methods return objects (other than strings) and are actually used in specifications
  - 1 out of 182 pure methods returns a newly allocated objects (besides strings)

## Problem 3: Frame Properties

- Model field solution does not work for methods with parameters
- Caching of values not possible for runtime checking
- Mutable dependent principle too strict

```
class List {
  pure bool Has( object o )
  { ... }
  void Remove( object o )
  requires Has( o );
  { ... }
  ... }

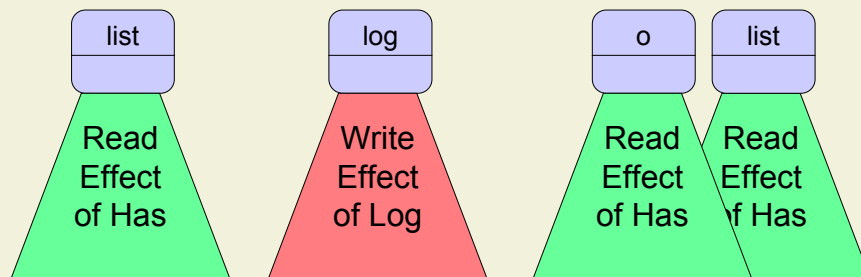
```

```
void Foo( List list, object o )
requires list.Has( o );
{
  log.Log( "Message" );
  list.Remove( o );
}

```

## Future Work: Non-Interference

- Possible solution: read and write effects
  - Data abstraction via ownership



## Summary

- Data abstraction is crucial to express functional correctness properties
- Verification methodology for model fields
  - Supports subtyping
  - Is modular and sound
  - Key insight: model fields are reduced to ordinary fields with automatic updates
- Verification methodology for methods
  - Partial solution: encoding, weak purity, consistency
  - Future work: frame properties based on effects