Verification of Object-Oriented Programs with Invariants

Mike Barnett, Robert DeLine, Manual Fahndrich, K. Rustan M. Leino an Wolfram Shulte

Overview

- **Goal**: design a sound methodology for specifying *object invariants* that can then be automatically verified (statically or dynamically)

- Object invariants describe a programmer intentions

Design by Contract

- Routine specifications describe a contract between a program and clients of that program
- Postconditions on constructors
- Pre and postconditons on methods
- Modifies clauses
  - All methods can modify newly allocated fields

Common View

- Callers need not be concerned with establishing preconditions of class T provided:
  - Fields are only modified within methods of T
  - Invariants established in postconditions of methods
- What's the problem?

Invariants May be Temporarily Violated!

```
class T{
  private x, y: int ;
  invariant 0 ≤ x < y;
  public T ( )
    {
          x = 0; y = 1;
    }
  public method M ( )
    modifies x, y;
    {
          x=x+3;
          P();
          y=4*y;
    }
  public method P ( )
    {
          M();
    }
}
```

Invariant violated: x=3, y=1

Include Explicit Pre-conditions?

```
class T{
  private x, y: int ;
  invariant 0 ≤ x < y;
  public T ( )
    {
        x = 0; y = 1;
    }
  public method M ( )
    requires 0 ≤ x < y;
    modifies x, y;
    {
        x=x+3;
        P();
        y=4*y;
    }
  public method P ( )
    {
        M();
    }
}
```

Exposes internal fields! Bad information hiding practices.

Proposed Solution

- Each object gets a special public field $st = \{Invalid, Valid\}$
  - If $o.st = Valid$, o's invariant is known to hold
  - If $o.st = Invalid$, o's invariant is not known to hold
- $Inv_T(o)$ holds ≡ the invariant declared in T holds for o (within a state)

Proposed Solution

- Fields can only be modified between ***unpack*** and ***pack*** statements

$$
\begin{aligned}
\text{pack } o \quad &\equiv \quad \text{assert } o \neq \mathit{null} \wedge o.st = \mathit{Invalid} \;; \\
&\qquad \text{assert } \mathit{Inv}_T(o) \;; \\
&\qquad o.st := \mathit{Valid} \\
\text{unpack } o \quad &\equiv \quad \text{assert } o \neq \mathit{null} \wedge o.st = \mathit{Valid} \;; \\
&\qquad o.st := \mathit{Invalid}
\end{aligned}
$$

Back to Our Example

```
class T{
   private x, y: int ;
   invariant 0 ≤ x < y;
   public T( )
     ensures st = Valid;
     {
           x = 0; y = 1;
           pack this;
     }
}
```

Postcondition

Precondition

```
public method M ( )
   requires st = Valid;
   modifies x, y;
   {
        unpack this;
        x=x+3;
        P();
        y=4*y;
        pack this;
   }
public method P ( )
   {
        M();
   }
}
```

Back to Our Example

```
class T{
  private x, y: int ;
  invariant 0 ≤ x < y;
  public T( )
    ensures st = Valid;
    {
          x = 0; y = 1;
          pack this;
    }
}
```

```
public method M( )
    requires st = Valid;
    modifies x, y;
    {
          unpack this;
          x=x+3;
          P();
          y=4*y;
          pack this;
    }
public method P( )
    {
          M();
    }
}
```

Modifies still exposes some fields to the client.

Why Not Just Check Invariant?

```
class T{
  private x, y: int ;
  invariant 0 ≤ x < y;
  public method M ( )
    requires st = Valid;
    modifies x, y;
    {
          …
          unpack this;
          x=x+3;
          y=4*y;
          pack this;
          …
     }
}
```

```
class T{
  private x, y: int ;
  invariant 0 ≤ x < y;
  public method M ( )
    modifies x, y;
    {
          checkInv ( );
          …
          x=x+3;
          y=4*y;
          …
          checkInv ( );
     }
  public method checkInv( )
  {
          assert ( 0 ≤ x < y );
   }
}
```

We Can Prove a Program Invariant

- If
  - field updates are only allowed when $o.st$ is invalid (i.e., between ***pack*** and ***unpack***)
  - we only allow the invariant to depend on fields of this (for now)
- Then

$$(\forall\, o : T \bullet\ o.st = \mathit{Invalid} \ \vee\ \mathit{Inv}_T(o) \ )$$

Extending to Components

```
class T{
  private f: U ;
  invariant 0 ≤ f.g;
  …
  public method M ( )
    requires st = Valid;
    {
          f.N ( ) ;
    }
   …
}
```

```
class U{
  private g: int ;
  …
  public method N( )
    requires st = Valid;
    {
          unpack this;
          g = -1 ;
          pack this;
    }
   …
}
```

T's invariant violated in a *Valid* state!

Include *f.st* in Precondition of *T*?

```
class T{
  private f: U ;
  invariant 0 ≤ f.g;
  …
  public method M ( )
    requires st = Valid;
    requires f.st = Valid;
    {
        unpack this;
        f.N ( ) ;
        pack this;
    }
  …
}
```

```
class U{
  private g: int ;
  …
  public method N( )
    requires st = Valid;
    {
        unpack this;
        g = -1 ;
        pack this;
    }
  …
}
```

Bad information hiding!

Solution?

- Prevent a class from being unpacked without regard to a class that might refer to it.

- t refers to u, so ***commit*** u to t

Committing

- Components identified with **rep** modifier
- $st = \{Valid, Invalid, Committed\}$

$$
\begin{aligned}
\textbf{pack}\ o \quad \equiv \quad & \text{assert}\ o \neq null \wedge o.st = Invalid\ ; \\
& \text{assert}\ Inv_T(o)\ ; \\
& \text{foreach}\ p \in Comp_T(o)\ \{\ \text{assert}\ p = null \vee p.st = Valid\ ;\ \} \\
& \text{foreach}\ p \in Comp_T(o)\ \{\ \text{if}\ (p \neq null)\ \{\ p.st := Committed\ ;\ \}\} \\
& o.st := Valid \\
\textbf{unpack}\ o \quad \equiv \quad & \text{assert}\ o \neq null \wedge o.st = Valid\ ; \\
& o.st := Invalid\ ; \\
& \text{foreach}\ p \in Comp_T(o)\ \{\ \text{if}\ (p \neq null)\ \{\ p.st := Valid\ ;\ \}\}
\end{aligned}
$$

Back to Our Example

```
class T{
  private rep f: U ;
  invariant 0 ≤ f.g;
  public T ( )
  {
        f.g = 10;
        pack this;
  }
  public method M ( )
    requires st = Valid;
    {
        unpack this;
        f.N ( ) ;
        pack this;
    }
   …
}
```

```
class U{
  private g: int ;
  …
  public method N( )
    requires st = Valid;
    {
        unpack this;
        g = -1 ;
        pack this;
    }
```

Commits *u* to *t*

Takes *t* from *Committed* to *Valid*

So what?

- If
  - field updates are only allowed when $o.st$ is invalid (i.e., between ***pack*** and ***unpack***)
  - object invariant can depend on fields of this and component fields declared with **rep** ($this.f_1.f_2\ldots.g$)
- Then
  - We can prove a stronger ***program invariant***:

$$(\forall\, o: T \bullet\ o.st = Invalid \vee (Inv_T(o) \wedge (\forall\, p \in Comp_T(o) \bullet\ p = null \vee p.st = Committed\ )))$$

Proving Program Invariant

- Requires all committed object have unique *owners*
- Can transfer owners from t to u via:

$$
\begin{array}{l}
\textbf{unpack}\ t\ ;\ \textbf{unpack}\ u\ ; \\
u.g := t.f\ ;\ \textbf{pack}\ u\ ; \\
t.f := \mathit{null}\ ;\ \textbf{pack}\ t\ ;
\end{array}
$$

Still Too Restrictive!

- If
  - field updates are only allowed when $o.st$ is invalid (i.e., between ***pack*** and ***unpack***
  - object invariant can depend on fields of this and component fields declared with **rep** ($this.f_1.f_2\ldots.g$)
- Then
  - We can prove a stronger ***program invariant***:

$$(\forall\, o{:}\, T \bullet\ o.st = Invalid \lor$$
$$(Inv_T(o) \land (\forall\, p \in Comp_T(o) \bullet\ p = null \lor p.st = Committed\ ))\ )$$

Subclasses

- Problem
  - o: B
  - class frame
    - Possible sets:
      - {object}
      - {object, A}
      - {object, A, B}

```
class object {        // pre-declared by the language
   // various declarations...
}
class A extends object {
   w: W ; x: X ;
   invariant ... w ... x ... ;
   // routine declarations...
}
class B extends A {
   y: Y ; z: Z ;
   invariant ... w ... x ... y ... z ... ;
   // routine declarations...
}
```

Object	Y	Y	Y	Y	N	N	N	N
A	Y	Y	N	N	Y	Y	N	N
B	Y	N	N	Y	Y	N	Y	N

Specifying them is enough

Subclasses

- Solution
  - Abandon ***st*** field
  - Introduce fields
    - ***inv***: the most derived class whose class frame is valid
    - ***committed***: boolean that indicates whether the object is committed

Subclasses

- Example

```
class Reader {
  public Reader()
    ensures inv = Reader ∧ ¬committed;
  public method GetChar(): int
    requires inv = 1 ∧ ¬committed;
    modifies this.{1} ;
    ensures −1 ⩽ result < 65536 ;
  ...
}
```

Replace “st” statement

Subclasses

- pack and unpack

Abandon st
Introduce inv, committed

$$
\begin{aligned}
&\textbf{pack } o \textbf{ as } T \equiv \\
&\quad \textbf{assert } o \neq null \wedge o.inv = S \,; \\
&\quad \textbf{assert } Inv_T(o) \,; \\
&\quad \textbf{foreach } p \in Comp_T(o) \,\{ \\
&\quad\quad \textbf{assert } p = null \vee (p.inv = \textbf{type}(p) \wedge \neg p.committed) \,; \,\} \\
&\quad \textbf{foreach } p \in Comp_T(o) \,\{ \textbf{ if } (p \neq null) \,\{ \, p.committed := true \,; \,\}\} \\
&\quad o.inv := T
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{unpack } o \textbf{ from } T \equiv \\
&\quad \textbf{assert } o \neq null \wedge o.inv = T \wedge \neg o.committed \,; \\
&\quad o.inv := S \,; \\
&\quad \textbf{foreach } p \in Comp_T(o) \,\{ \textbf{ if } (p \neq null) \,\{ \, p.committed := false \,; \,\}\}
\end{aligned}
$$

Routine specifications

- What is routine specification?
  - A contract between its callers and implementations, which describes what is expected of the caller at the time of call, and what is expected of the implementation at the time of return.

Routine specifications

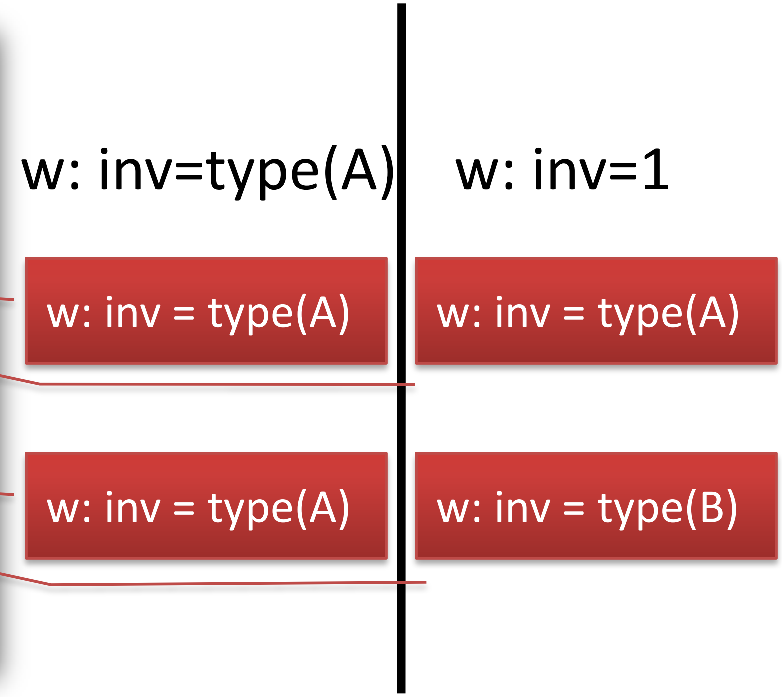
- Writing modifies clauses
  - Definitions
    - o: object
    - f: field name of o
    - Heap[o, f]:
    - W: modifies clause
  - Policy

$$
\begin{aligned}
(\forall\, o, f \bullet\; & Heap[o,f] = \mathbf{old}(Heap[o,f]) \lor (o,f) \in \mathbf{old}(W) \lor \\
& \neg\mathbf{old}(Heap[o, alloc]) \lor \mathbf{old}(Heap[o, committed])\;)
\end{aligned}
$$

Routine specifications

- Writing preconditions of methods and overrides
  - Dynamically dispatched method
  - Define 1 as type(this)

```
class object {        // pre-declared by the language
   // various declarations...
}
class A extends object {
   w: W ;  x: X ;
   invariant ... w ... x ... ;
   // routine declarations...
}
class B extends A {
   y: Y ;  z: Z ;
   invariant ... w ... x ... y ... z ... ;
   // routine declarations...
}
```



Example - readers

```
class Reader {
  public Reader( )
    ensures inv = Reader ∧ ¬committed ;
  public method GetChar( ): int
    requires inv = 1 ∧ ¬committed ;
    modifies this.{1} ;
    ensures −1 ⩽ result < 65536 ;
  . . .
}
```

By Default

Not committed to anyone else

Example – array readers

$$
\begin{array}{l}
\textbf{class}\ \mathit{Reader}\ \{ \\
\quad \textbf{public}\ \mathit{Reader}(\,) \\
\quad\quad \textbf{ensures}\ \mathit{inv} = \mathit{Reader} \wedge \neg\mathit{committed}\ ; \\
\quad \textbf{public method}\ \mathit{GetChar}(\,)\colon \textbf{int} \\
\quad\quad \textbf{requires}\ \mathit{inv} = 1 \wedge \neg\mathit{committed}\ ; \\
\quad\quad \textbf{modifies}\ \mathit{this}.\{1\}\ ; \\
\quad\quad \textbf{ensures}\ -1 \leqslant \textbf{result} < 65536\ ; \\
\quad \ldots \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{ArrayReader}\ \textbf{extends}\ \mathit{Reader}\ \{ \\
\quad \textbf{private rep}\ \mathit{src}\colon \textbf{char}[\,]\ ; \\
\quad \textbf{private}\ n\colon \textbf{int}\ ; \\
\quad \textbf{invariant}\ 0 \leqslant n \leqslant \mathit{src.length}\ ; \\
\quad \textbf{public}\ \mathit{ArrayReader}(\mathit{source}\colon \textbf{char}[\,]) \\
\quad\quad \textbf{requires}\ \mathit{source} \neq \mathit{null} \wedge \mathit{source.inv} = \textbf{type}(\mathit{source}) \wedge \neg\mathit{source.committed}\ ; \\
\quad\quad \textbf{ensures}\ \mathit{inv} = \mathit{ArrayReader} \wedge \neg\mathit{committed}\ ; \\
\quad\quad \ldots \\
\quad \textbf{impl}\ \mathit{GetChar}(\,)\colon \textbf{int}\ \{ \\
\quad\quad \textbf{var}\ \mathit{ch}\colon \textbf{int}\ ; \\
\quad\quad \textbf{unpack}\ \mathit{this}\ \textbf{from}\ \mathit{ArrayReader}\ ; \\
\quad\quad \textbf{if}\ (n = \mathit{src.length})\ \{\ \mathit{ch} := -1\ ;\ \} \\
\quad\quad \textbf{else}\ \{\ \mathit{ch} := (\textbf{int})\mathit{src}[n]\ ;\ \ n := n + 1\ ;\ \} \\
\quad\quad \textbf{pack}\ \mathit{this}\ \textbf{as}\ \mathit{ArrayReader}\ ; \\
\quad\quad \textbf{return}\ \mathit{ch}\ ; \\
\quad \} \\
\}
\end{array}
$$

inv = type(Reader)
this.{type(Reader)}

inv = type(ArrayReader)
this.{type(ArrayReader)}

Example – parameter passing

```
class ArrayReader extends Reader {
  private rep src: char[] ;
  private n: int ;
  invariant 0 ⩽ n ⩽ src.length ;
  public ArrayReader(source: char[])
    requires source ≠ null ∧ source.inv = type(source) ∧ ¬source.committed ;
    ensures inv = ArrayReader ∧ ¬committed ;
  { super() ;
    src := source ; n := 0 ;
    pack this as ArrayReader ;
  }

  impl GetChar(): int {
    var ch: int ;
    unpack this from ArrayReader ;
    if (n = src.length) { ch := −1 ; }
     else { ch := (int)src[n] ; n := n + 1 ; }
    pack this as ArrayReader ;
    return ch ;
  }
}
```

source.committed goes from *false* to *true* violating the precondition

Now What?

```
class ArraySort  { // Insertion Sort Method by  R. Monahan & R. Leino / APH
 public static void sortArray( int[]! a )
   modifies a[*];
   ensures forall{int j in (1:a.Length);(a[j-1] <= a[j])};
 {
   int  t, k=1;
   if (a.Length > 0)  {
      while(k < a.Length)
         invariant  1 <= k && k <= a.Length;
         invariant  forall { int j in (1:k), int i in (0:j); (a[i] <= a[j]) };
     {
         for( t = k;  t>0 && a[t-1]>a[t];  t-- )
            invariant   k < a.Length;
            invariant   0<=t && t<=k;
            invariant forall {  int j in (1:k+1), int i in (0:j);  j==t || a[i] <= a[j] };
         {  int temp;  temp = a[t];  a[t] = a[t-1];  a[t-1] = temp; }
         k++;
    } } }
```

Spec#

- Specifications integrated into Spec# which extends C#
- Spec# compiler integrated into Visual Studio
- Boogie statically verifies correctness and finds errors

Thanks!