Verification of Object-Oriented Programs with Invariants

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 postconditions 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 \( \equiv \) the invariant declared in \( T \)
  holds for \( o \) (within a state)
Proposed Solution

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

\[
\begin{align*}
\text{pack } o & \equiv \text{assert } o \neq \text{null } \land o.st = \text{Invalid } ; \\
& \quad \text{assert } Inv_T(o) ; \\
& \quad o.st := \text{Valid} \\
\text{unpack } o & \equiv \text{assert } o \neq \text{null } \land o.st = \text{Valid } ; \\
& \quad o.st := \text{Invalid}
\end{align*}
\]
Back to Our Example

class T{
    private x, y: int;
    invariant 0 \leq 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();
    }
}
class $T$
  
  private $x$, $y$: int;
  
  invariant $0 \leq x < y$;

public $T()$
  
  ensures $st = Valid$;
  
  {
    $x = 0$; $y = 1$;
    pack this;
  }

}

public method $M()$

  requires $st = Valid$;
  modifies $x$, $y$;
  
  {
    unpack this;
    $x = x + 3$;
    $P()$;
    $y = 4 \times y$;
    pack this;
  }

public method $P()$

  {
    $M()$;
  }

Modifies still exposes some fields to the client.
class $T$
{
    private $x$, $y$: int;
    invariant $0 \leq x < y$;
    public method $M( )$
    requires $st = Valid$;
    modifies $x$, $y$;
    {
        ...
        unpack this;
        $x$ = $x$ + 3;
        $y$ = $4 \times y$;
        pack this;
        ...
    }
    ...
    public method checkInv( )
    {
        assert ($0 \leq x < y$);
    }
}
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

\[
(\forall o : T \bullet o.st = \text{Invalid} \lor \text{Inv}_T(o))
\]
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 $0 \leq f.g$;
    ...
    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
- \( st = \{ \text{Valid, Invalid, Committed} \} \)

\[
\begin{align*}
\text{pack } o & \equiv \text{assert } o \neq \text{null } \land o.st = \text{Invalid} ; \\
& \quad \text{assert } \text{Inv}_T(o) ; \\
& \quad \text{foreach } p \in \text{Comp}_T(o) \{ \text{assert } p = \text{null } \lor p.st = \text{Valid} ; \} \\
& \quad \text{foreach } p \in \text{Comp}_T(o) \{ \text{if } (p \neq \text{null}) \{ p.st := \text{Committed} ; \} \} \\
& \quad o.st := \text{Valid} \\
\text{unpack } o & \equiv \text{assert } o \neq \text{null } \land o.st = \text{Valid} ; \\
& \quad o.st := \text{Invalid} ; \\
& \quad \text{foreach } p \in \text{Comp}_T(o) \{ \text{if } (p \neq \text{null}) \{ p.st := \text{Valid} ; \} \}
\end{align*}
\]
Back to Our Example

class \( T \{ \)
  private \( \text{rep} \); \( U \); 
  invariant \( 0 \leq f.g \); 
  public \( T() \) 
  { 
    \( f.g = 10; \)
    \( \text{pack this;} \)
  }
  public method \( M() \) 
    requires \( st = \text{Valid} \); 
  { 
    \( \text{unpack this;} \)
    \( f.N() ; \)
    \( \text{pack this;} \)
  }
  ... 
}

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

Commits \( u \) to \( t \)
Takes \( t \) from \( \text{Committed to Valid} \)
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 $\text{rep}(this.f_1.f_2....g)$

• Then
  – We can prove a stronger program invariant:

\[
(\forall o: T \bullet o.st = \text{Invalid} \lor \\
(\forall p \in \text{Comp}_T(o) \bullet p = \text{null} \lor p.st = \text{Committed}))
\]
Proving Program Invariant

• Requires all committed object have unique owners
• Can transfer owners from $t$ to $u$ via:

\[
\text{unpack } t ; \text{unpack } u ; \\
u.g := t.f ; \text{pack } u ; \\
t.f := \text{null} ; \text{pack } t ;
\]
Still Too Restrictive!

• If
  – field updates are only allowed when \( o.st \) is invalid (i.e., between \textit{pack} and \textit{unpack}.
  – object invariant can depend on fields of this and component fields declared with \texttt{rep \((\texttt{this.f1.f2} \ldots \texttt{g})\))}

• Then
  – We can prove a stronger \textit{program invariant}:

\[
(\forall o : T \bullet o.st = \text{Invalid} \lor \\
(Inv_T(o) \land (\forall p \in \text{Comp}_T(o) \bullet p = \text{null} \lor p.st = \text{Committed})))
\]
Subclasses

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

Object | Y | Y | Y | Y | Y | N | N | N | N | N
A      | Y | Y | N | N | N | Y | Y | N | N | N
B      | Y | N | N | N | Y | Y | N | Y | N | 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;
        ...
    }
}
Subclasses

- pack and unpack

pack $o$ as $T$ ≡

assert $o \neq \text{null} \land o.inv = S$;
assert $\text{Inv}_T(o)$;
foreach $p \in \text{Comp}_T(o)$ {
    assert $p = \text{null} \lor (p.inv = \text{type}(p) \land \neg p\text{.committed})$;
}
foreach $p \in \text{Comp}_T(o)$ {
    if ($p \neq \text{null}$) {
        $p\text{.committed} := \text{true}$;
    }
}  
$o.inv := T$

unpack $o$ from $T$ ≡

assert $o \neq \text{null} \land o.inv = T \land \neg o\text{.committed}$;
$o.inv := S$;
foreach $p \in \text{Comp}_T(o)$ {
    if ($p \neq \text{null}$) {
        $p\text{.committed} := \text{false}$;
    }
}
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

\[(\forall o, f \cdot Heap[o, f] = \text{old}(Heap[o, f]) \lor (o, f) \in \text{old}(W) \lor \neg \text{old}(Heap[o, alloc]) \lor \text{old}(\text{Heap}[o, \text{committed}]) )\]
Routine specifications

• Writing preconditions of methods and overrides
  – Dynamically dispatched method
  – Define 1 as `type(this)`
Example - readers

```java
class Reader {
    public Reader() {
        ensures inv = Reader \land \neg \text{committed} ;
    }
    public method GetChar(): int {
        requires inv = 1 \land \neg \text{committed} ;
        modifies this.{1} ;
        ensures -1 \leq result < 65536 ;
    }
    ...
}
```
Example – array readers

```java
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;
    }
    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;
    }
}
```
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!