A Fault Model for Subtype Inheritance and Polymorphism

Jeff Offutt
Information & Software Engineering
George Mason University
Fairfax, VA USA
www.ise.gmu.edu/faculty/ofut/
ofut@ise.gmu.edu

Joint research with: Roger T. Alexander, Colorado State University
Ye Wu, George Mason University
Quansheng Xiao, George Mason University
Chuck Hutchinson, George Mason University

Supported by NSF and NIST.

Inheritance

Allows common features of many classes to be defined in one class

A derived class has everything its parent has, plus it can:

• **Enhance** derived features (overriding)
• **Restrict** derived features
• **Add** new features (extension)

(C) Copyright 1998-2001. All Rights Reserved.
Inheritance (2)

**Declared type:** The type given when an object reference is declared
Clock w1; // declared type Clock

**Actual type:** The type of the current object
w1 = new Watch(); // actual type Watch

In Java, the method that is executed is the lowest version of the method defined between the actual and root types in the inheritance hierarchy

Polymorphism

• The same variable can have different types depending on the program execution

• If B inherits from A, then an object of type B can be used when an object of type A is expected

• If both A and B define the same method $m()$ ($B$ overrides $A$), then the same statement will sometimes call $A$’s version of $M$, and sometimes $B$’s version
Subtype and Subclass Inheritance

- **Subtype Inheritance**: If B inherits from A, any object of type B can be substituted for an object of type A
  - A laptop “is a” special type of computer
  - Called substitutability

- **Subclass Inheritance**: Objects of type B may not be substituted for objects of type A
  - Objects of B may not be “type compatible”

This talk assumes subtype inheritance, subclass inheritance will be addressed later.

Example

<table>
<thead>
<tr>
<th>Method</th>
<th>Defs</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A::h</td>
<td>{A::u,A::w}</td>
<td></td>
</tr>
<tr>
<td>A::i</td>
<td>{A::v}</td>
<td></td>
</tr>
<tr>
<td>A::j</td>
<td>{A::v}</td>
<td></td>
</tr>
<tr>
<td>A::l</td>
<td>{A::v}</td>
<td></td>
</tr>
<tr>
<td>B::h</td>
<td>{B::x}</td>
<td></td>
</tr>
<tr>
<td>B::i</td>
<td>{B::x}</td>
<td></td>
</tr>
<tr>
<td>C::i</td>
<td>{C::y}</td>
<td></td>
</tr>
<tr>
<td>C::j</td>
<td>{C::y}</td>
<td></td>
</tr>
<tr>
<td>C::l</td>
<td>{A::v}</td>
<td></td>
</tr>
</tbody>
</table>

(C) Copyright 1998-2001. All Rights Reserved.
Polymorphism Headaches (Yo-Yo)

Object is of type A
A::d ()
Polymorphism Headaches (Yo-Yo)

Object is of type C, C::d ()

Potential for Faults in OO Programs

• Complexity is relocated to the connections among components
• Less static determinism – many faults can now only be detected at runtime
• Inheritance and Polymorphism yield vertical and dynamic integration
• Aggregation and use relationships are more complex
• Designers do not carefully consider visibility of data and methods
A Fault-Failure Model for OO Programs

1. **Reachability**: The faulty location must be reached
   1. class $D$ *extends* $T$
   2. $D$ has a method $m()$ that *overrides* $T$’s $m()$
   3. There is an object $o$ of type $T$ ($T\ o;$)
   4. The actual type of $o$ is $D$ ($o = $new $D();$)
   5. $m()$ is called in the context of $o$ ($o.m();$)

2. **Infection**: Program state must be incorrect
   - $T::m()$ and $D::m()$ modify different variables; both are available to $T$
     ($T::m()$ defines $x$ and $D::m()$ defines $y$)

3. **Propagation**: Program output must be incorrect
   - A path exists from the definition of the variable in $T::m()$ or $D::m()$ to a use
   - The use of the variable affects the output state of the program

Object-oriented Faults

- Only consider faults that arise as a direct result of OO language features:
  - inheritance
  - polymorphism
  - constructors
  - visibility
- Language independent (as much as possible)
OO Faults and Anomalies

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Fault / Anomaly</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITU</td>
<td>Inconsistent Type Use</td>
</tr>
<tr>
<td>SDA</td>
<td>State Definition Anomaly</td>
</tr>
<tr>
<td>SDIH</td>
<td>State Definition Inconsistency</td>
</tr>
<tr>
<td>SDI</td>
<td>State Defined Incorrectly</td>
</tr>
<tr>
<td>IISD</td>
<td>Indirect Inconsistent State Definition</td>
</tr>
<tr>
<td>ACB1</td>
<td>Anomalous Construction Behavior (1)</td>
</tr>
<tr>
<td>ACB2</td>
<td>Anomalous Construction Behavior (2)</td>
</tr>
<tr>
<td>IC</td>
<td>Incomplete Construction</td>
</tr>
<tr>
<td>SVA</td>
<td>State Visibility Anomaly</td>
</tr>
</tbody>
</table>

Inconsistent Type Use (ITU)

- No overriding (no polymorphism)
- $C$ extends $T$, and $C$ adds new methods (extension)
- An object is used "as a $T$", then as a $C$, then as a $T$
- Methods in $T$ can put object in a state that is inconsistent for $C$

```java
Vector
-array
+insertElementAt()
+removeElementAt()
```

```java
Stack
+pop (): Object
+push (): Object
```

```java
void dumb (Vector v)
{
    v.removeElementAt (v.size()-1);
}
```

```java
s.push ("steffi");
s.push ("joyce");
s.push ("chelsea");
dumb (s);
s.pop();
s.pop();
s.pop();  // Stack is empty!
```
State Definition Anomaly (SDA)

- \( C \) extends \( T \), and \( C \) overrides some methods
- The overriding methods in \( C \) fail to define some variables that the overridden methods in \( T \) defined

\[
\begin{array}{c}
W \\
v \\
m() \\
n() \\
X \\
x \\
n() \\
Y \\
w \\
m()
\end{array}
\]

- \( W::m() \) defines \( v \) and \( W::n() \) uses \( v \)
- \( X::n() \) uses \( v \)
- \( Y::m() \) does not define \( v \)

For an object of type \( Y \), a data flow anomaly exists and results in a fault if \( m() \) is called, then \( n() \).

State Definition Inconsistency (SDIH)

- Overriding a variable, possibly accidentally
- If the descendant’s version of the variable is defined, the ancestor’s version may not

\[
\begin{array}{c}
W \\
v \\
m() \\
n() \\
X \\
x \\
n() \\
Y \\
v \\
m()
\end{array}
\]

- \( Y \) overrides \( W \)'s version of \( v \)
- \( Y::m() \) defines \( Y::v \)
- \( X::n() \) uses \( v \)

For an object of type \( Y \), a data flow anomaly exists and results in a fault if \( m() \) is called, then \( n() \).
State Defined Incorrectly (SDI)

- Overriding a method $m()$ that defines a variable $v$
- The overriding method may define $v$ incorrectly

![Diagram of SDI](image1)

- $W::n()$ defines $v$
- $X::n()$ also defines $v$, but incorrectly

For an object of type $X$, a behavioral error exists occurs if $W::m()$ uses $v$ and assumes it has a value as given in $W::n()$

Indirect Inconsistent State Definition (IISD)

- A method is added that defines an inherited state variable
- Method puts the ancestor in an inconsistent state

![Diagram of IISD](image2)

- $W::m()$ cannot call $X::e()$
- $X::m()$ calls $X::e()$, which defines $W::y$ incorrectly …
Anomalous Construction Behavior (ACB1)

- Constructor of C calls a method f()
- Child of C, D, overrides f()
- D::f() uses variables that should be defined by D’s constructor, but are not

<table>
<thead>
<tr>
<th>Calls</th>
<th>Overrides</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f()</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f()</td>
<td></td>
</tr>
</tbody>
</table>

When an object of type D is constructed, C() is run before D().

When C() calls D::f(), x is used, but has not yet been given a value!

Anomalous Construction Behavior (ACB2)

- Constructor of C calls a method f()
- Child of C, D, overrides f()
- D::f() uses variables that are defined by C’s constructor

<table>
<thead>
<tr>
<th>Calls</th>
<th>Overrides</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f()</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f()</td>
<td></td>
</tr>
</tbody>
</table>

The author of C cannot know anything about D::f()

If D::f() uses a variable x that C() defines, and the definition is after the call to f(), x has no value in D::f()
Incomplete Construction (IC)

- Constructors should give all variables “reasonable” values
- In C++, the variables have no values by default!
- Two possible faults:
  1. Wrong value assigned to a variable
  2. No value assigned to a variable (more dangerous in C++)

State Visibility Anomaly (SVA)

- A private variable v is defined in ancestor A, and v is defined by A::m()
- B extends A and C extends B
- C overrides m(), and calls A::m() to define v

```
A
  -v
  m()

B
  m()

C
  m()

A
  -v
  m()

B
  m()

C
  m()
```

B::m() is added later
C::m() can no longer call A::m()!
Conclusions

- A model for understanding and analyzing faults that occur as a result of inheritance and polymorphism
- Technique for identifying data flow anomalies in class hierarchies
- Guidelines for proper use of inheritance, polymorphism, and constructors

Future Work

- How often do these faults occur in practice
- Fault injection techniques for OO experimentation
- Guidelines for developing safe inheritance hierarchies
- Guidelines or standards for safe use of polymorphism
  - It is unsafe for constructors to call polymorphic methods
- Mutation operators for Java