

Quality Assurance II

Outline

These slides will be covered during TWO lectures.

Introduction

Formal Verification

Theoretical Limits

Practical Implications

Software Quality

Previous discussions relating to QA have focused on


- process quality
 - * related to the currently popular Demming notions
 - * manifested in QA plans
- testing
 - * most important technical part of QA
 - * testing has two levels – strategy and tactics
- metrics
 - * error tracking & prediction

Software Quality Assurance

Process

- methods and tools
- configuration management and change control
- standards and standard compliance mechanisms
- reporting mechanisms

Verification and Validation

- formal technical reviews
- multi-tiered testing strategy
- measurement
- symbolic execution
-  formal verification techniques

Verification *versus* Validation

Verification:

“Are we building the product right?”

◇ implementation meets specifications

Validation:

“Are we building the right product?”

◇ specifications meet needs

Symbolic Execution

Goal: reason about history of value changes

Represent

- values with character strings
- operations by augmented strings

≈ trace

For example:

```
readf "%d", x;  
y := x + 2;  
z := y + 3;
```

sets

```
x    ←    "input"  
y    ←    "input + 2"  
z    ←    "(input + 2) + 3"
```

- expression simplification

Formal Verification

Program Proving

- vitally dependent on specification
formal verification can only be done against a formal specification

Techniques

- assertions
- formal semantics

In depth

- P415/P515 Introduction to Verification

Basic Idea

Each statement s is surrounded by logical formulae F_1 and F_2

Notation:

$$\{ F_1 \} s \{ F_2 \}$$

Interpretation:

In a correct program, if F_1 is true before s , then F_2 is true afterwards.

F_1 is called a *pre-condition* for s

F_2 is called a *post-condition* for s

Both are called *assertions*

Assertions

- for example

$$\{y > 5\} \quad x := y - 2 \quad \{x > 3 \ \& \ y > 5\}$$

- in a program:
 - always possible as comments
 - some languages provide ASSERT statements

```
ASSERT y > 5;  
x := y - 2;  
ASSERT x > 3 & y > 5;
```

- can “chain” assertions

$$\{F_1\} S_1 \{F_2\} S_2 \{F_3\}$$

Formal Deduction

Program with assertions are manipulated in a formal deduction system based on *proof rules*, such as

$$\frac{\{F_1\} S_1 \quad \{F_2\} , \quad \{F_2\} S_2 \quad \{F_3\}}{\{F_1\} S_1 ; S_2 \quad \{F_3\}}$$

These proof rules are in the same spirit as the proof rules of “classical” logic, such as

$$\frac{A , A \Rightarrow B}{B}$$

Assertions and Loops

Loop Invariant:

an assertion preserved by the body of a loop

If $\{F\}$ is invariant for a particular loop (body)
then $\{F\}$ is preserved by the entire loop

proof: by induction on number of iterations

Write $\{\{F\}\}$ to indicate F is a loop invariant

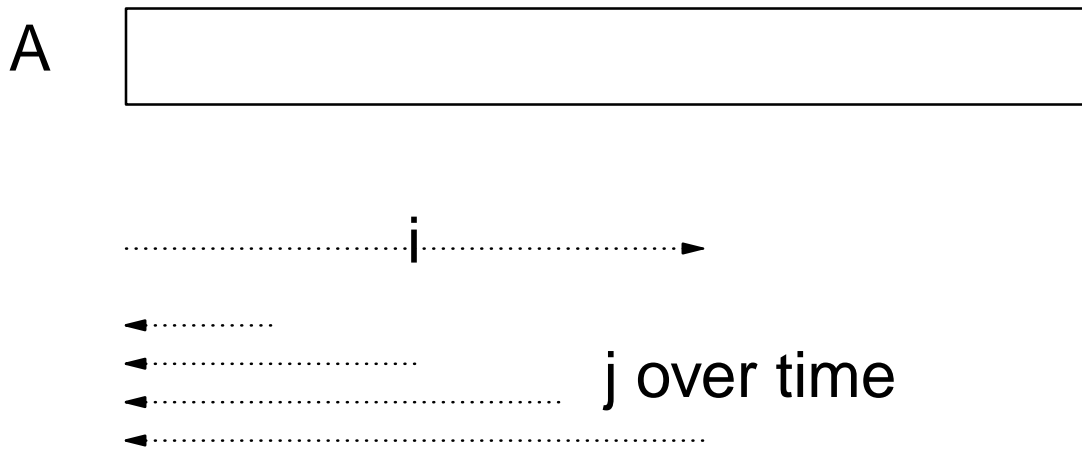
Example Procedure

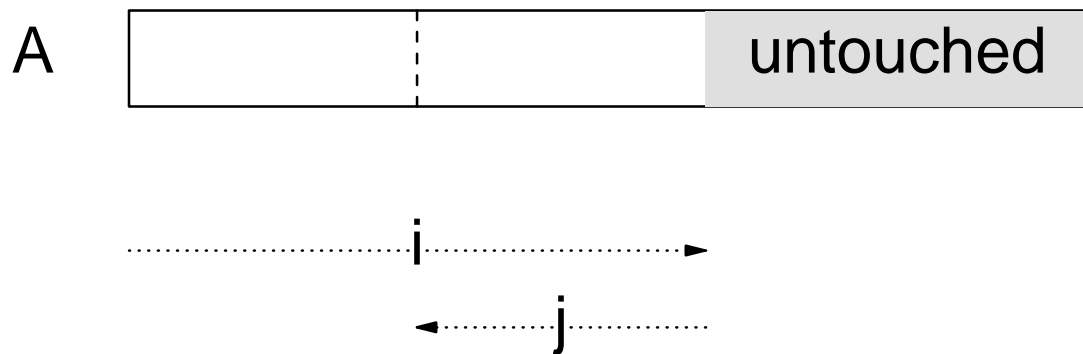
```
Procedure BubbleSort(  
  A: ARRAY OF INT, lim: INT );  
  for i := 2 to lim do  
    j := i;  
    while j>1 and A[j]<A[j-1] do  
      A[j] :=: A[j-1];  
      j := j-1  
    end  
  endfor;
```

Notation

$$X ::= Y$$

means interchange X and Y .





Procedure Pre- & Post-conditions

```

{ lim, A[1..lim] are integers }
Procedure BubbleSort(
  A: ARRAY OF INT, lim: INT );
  i := 2;
  while i ≤ lim do
    j := i;
    while j > 1 and A[j] < A[j-1] do
      A[j] := A[j-1];
      j := j-1
    end
    i := i+1;
  end;
{ values in A unchanged &
  lim, A[1..lim] are integers &
  i = lim + 1 &
  forall x, 1 < x ≤ lim :
    A[x-1] ≤ A[x] }

```

Also, the **for** loop rewritten into simpler components

Loop Invariants

```
Procedure BubbleSort(  
  A: ARRAY OF INT, lim: INT );  
  i := 2;  
  while i<=lim do  
    { A[1..i-1] sorted }  
    j := i;  
    while j>1 and A[j]<A[j-1] do  
      A[j] := A[j-1];  
      j := j-1  
    end  
    { A[1..i] sorted }  
    i := i+1;  
    { A[1..i-1] sorted }  
  end;  
{ i = lim+1 &  
  A[1..lim] sorted }
```

Note – certain less important assertions omitted

Loop Invariants

```

Procedure BubbleSort(
  A: ARRAY OF INT, lim: INT );
  i := 2; {A[1] sorted}
  {{ A[1..i-1] sorted }}
  while i<=lim do

    j := i;
    {{ A[1..j-1] sorted &
      A[j..i] sorted }}
    while j>1 and A[j]<A[j-1] do
      A[j] :=: A[j-1];
      j := j-1
    end
    { A[1..i] sorted }
    i := i+1;

  end;
  { i = lim+1 &
    A[1..lim] sorted }

```

Loop Invariant Formalism

Let I (intended as the loop invariant) and $cond$ (where $\sim cond$ is intended as the loop termination condition) be assertions.

The formal inference rule for a loop with a body B is:

$$\frac{\{ I \ \& \ cond \} B \{ I \}}{\{ I \} \mathbf{while} \ cond \ \mathbf{do} \ B \ \mathbf{end} \{ I \ \& \ \sim cond \}}$$

Invariant assertions only worthwhile if loop terminates

Termination

```
{ x > y }
{{ x > y }}
while x <> 0 do
    x := x-2 ;
    y := y-2 ;
end
{ x > y & x = 0 }
```

The proof rule shows the above to be correct

but the loop terminates

if and only if x is initially even and positive.

Loop Invariants and Induction

Tail recursion \Rightarrow iteration

```
f( i, x ) := if ( i=0 )  
  then g(x)  
  else h( i, f( i-1, x ) )
```

becomes

```
f( m, x ) {  
  r := g(x);  
  for i = m until 1 do  
    r := h( i, r );  
  return r }
```

Loop invariant on iterative version is very close to induction on recursive version

Loop Invariants as Design Aid

Formalism is not necessary

Ask:

- ★ What is changed in this loop?
- ★ What is preserved in this loop?

Look for higher-level concepts/properties
(such as part of array being sorted)

Back to Termination

It would be valuable to have a method to discover whether or not a program terminates.

Why not just run it?

- ☆ Fundamental limitation to computer science

The Halting Problem

(A *problem* is a set of inputs which we would like to classify by some program.)

Q: Can we tell whether or not an arbitrary program halts?

A: In general, no!

Intuition for the proof:

This statement is false.

true \Rightarrow **false**
false \Rightarrow **true**

This contradiction requires:

- self-reference
- negation

Statement & Proof Outline

Theorem: It is impossible to write a procedure that decides whether an arbitrary program halts.

Outline: proof by contradiction

1. assume the result is false (*i.e.* such a procedure does exist)
 2. define a new program using assumption
 3. consider cases when this new program halts
 - a. if it halts \Rightarrow it doesn't
 - b. if it doesn't \Rightarrow it does
- \therefore program cannot be constructed, so procedure does **not** exist.

Proof – Step 1

Assume there is a Boolean procedure `DoesHalt?` such that:

`DoesHalt?(Pfile)` returns

true if the program represented by `Pfile` halts and

false if the program goes into an infinite loop.

Preconditions:

- a. `Pfile` is a text file
- b. `Pfile` represents a syntactically correct program

Postconditions:

- a. `Pfile` is not changed
- b. The value true or false is returned as indicated above

Proof – Step 2

On the file *Confound* put the program:

```

/* file Confound */
main( )
#include DoesHalt?
{
    if (DoesHalt?( _____ ))
    {
        /* loop forever */
        while ( true ) { };
    }
}

```

(will fill in " _____ " later)

That is, the program Confound halts

if DoesHalt?(_____)
is FALSE

but it loops forever

if DoesHalt?(_____)
is TRUE.

A “Thought Experiment”

Pretend to execute the program by:

1. compiling the program file `Confound`
2. running the compiled program

That is:

```
%cc Confound
```

```
%a.out
```

What would happen???

Would it halt???

Proof – Step 3

Two cases: DoesHalt? in Confound

- either returns TRUE
- or it returns FALSE:

case 1 -

DoesHalt? (" Confound ") is TRUE

⇒

the program loops

⇒

DoesHalt? (" Confound ") is FALSE

case 2 -

DoesHalt? (" Confound ") is FALSE

⇒

the program halts

⇒

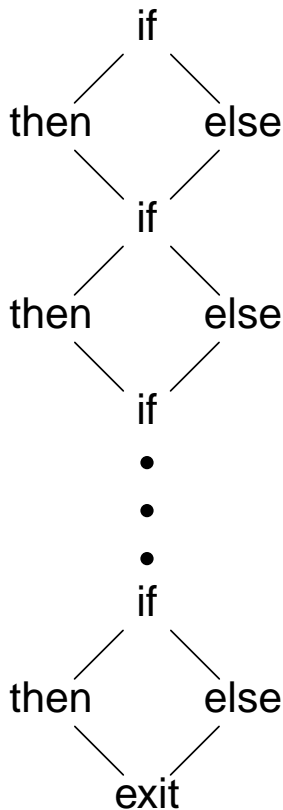
DoesHalt? (" Confound ") is TRUE

Consequences of Halting Problem

- halting problem takes on many “disguises”
- virtually any program that takes other programs as input is vulnerable:
 - ◇ compiler optimization
dead code elimination
 - ◇ program verification
 - ◇ concurrency synchronization
 - ◇ expert systems
- sometimes restrict power to avoid trap
 - ◇ SQL, other database query languages
- in general, a valuable warning

Testing and Formal Verification

Recall the complexity problem with testing a long chain of IF's:



n IF's, 2^n possible paths

exhaustive testing impossible

Proving that many conditions are mutually exclusive

⇒ many fewer possible paths

⇒ exhaustive tests may be possible

Proofs in Practice

- dependence on formal specifications
 \Rightarrow
cannot *prove* systems with complex interactions
- program proving useful for critical internal modules
 - **abstract data types (ADT's)**
 - **synchronizations**
- **proofs complement testing**
useful for hard-to-test situations

Proofs in Practice

Just as the typical mathematician does not prove theorems using the predicate calculus, the typical computer scientist should not prove programs by formal deduction.

Hence

- human-readable proofs
- “programmer’s assistant” to handle details
- CASE tools should facilitate use of assertions

Proof-like algorithms often internal

- ◇ type checking
- ◇ database validation

Example: Trusted Agents

Goal: admit “foreign” code inside of
“firewall”

Application: network protocol checkers
small, quick

Traditional approaches: interpretation,
bounds checking
essentially, a mini-firewall

Alternative: foreign module includes both
code & proof of proper behavior

- ◇ use the fact that proof checking is
easy even if proof discovery is
hard

Ten Commandments of Formal Methods

1. choose appropriate notation
2. formalize but don't over-formalize
3. estimate costs
4. have a formal methods expert on call
5. augment rather than replace traditional methods
6. document sufficiently
7. don't compromise other quality standards
8. do not be dogmatic
9. test, test, and test again
10. reuse

– after Bowan and Hinchley, *Computer* April 1995