Analysis of Software Artifacts

Hoare Logic: Proving Programs Correct

Jonathan Aldrich

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - Effective testing practices
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

What are the limits of testing?

- **What we can test**
  - Attributes that can be directly evaluated externally
    - *Examples:* **Functional** properties: result values, GUI manifestations, etc.
  - Attributes relating to resource use
    - Many well-distributed **performance** properties
    - Storage use
- **What is difficult to test?**
  - Attributes that **cannot easily be measured externally**
    - Is a design evolvable? — Inspection; Patterns; Design Structure Matrices
    - Is a design secure? — Secure Development Lifecycle; STRIDE
    - Is a design technically sound? — Model checking; Alloy; see also Models
    - Does the code conform to a design? — Plural (API usage); ArchJava; Reflexion models
    - Where are the performance bottlenecks? — Performance analysis; Profiling
    - Does the design meet the user's needs? — Usability analysis
  - Attributes for which **tests are nondeterministic**
    - Real time constraints — Rate monotonic scheduling
    - Race conditions — Analysis of locking
  - Attributes relating to the **absence of a property**
    - Absence of security exploits — Microsoft's Standard Annotation Language
    - Absence of memory leaks — Cyclone, Purify
    - Absence of functional errors — Hoare Logic; ESC/Java
    - Absence of non-termination — Termination analysis

Course Topics

- Classical quality assurance
  - Testing
  - Inspection
- Design analysis
  - Patterns
  - Frameworks
- Formal specification and verification
  - Hoare Logic: proving programs correct
  - ESC/Java: automated property checking
  - Plural: API usage verification
- Static analysis
  - Dataflow analysis
  - Model checking
  - Applications: Concurrency, security
- Special topics
  - Performance analysis
  - Security analysis
  - Reliability and defect prediction
  - Quality assurance in the organization: Microsoft, eBay, etc.

Testing and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now

- Proofs
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - May be practical for small programs in the future

- So why study proofs if they aren't (yet) practical?
  - Proofs tell us how to ***think*** about program correctness
    - Important for development, inspection
  - Foundation for static analysis tools
    - These are just simple, automated theorem provers
    - Many are practical today!

How would you argue that this program is correct?

```
/*@ requires len >= 0 && array.length == len
  @
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
      float sum = 0.0;
      int i = 0;
      while (i < len) {
            sum = sum + array[i];
            i = i + 1;
      }
      return sum;
}
```

Notation from the Java Modeling Language (JML)

Hoare Triples

- Formal reasoning about program correctness using pre- and postconditions

- Syntax: {P} S {Q}
  - P and Q are predicates
  - S is a program

- Semantics
  - If we start in a state where P is true and execute S, then S will terminate in a state where Q is true

Hoare Triple Examples

- { true } x := 5 { }
- { } x := x + 3 { x = y + 3 }
- { } x := x * 2 + 3 { x > 1 }
- { x=a } if (x < 0) then x := -x { }
- { false } x := 3 { }
- { x < 0 } while (x!=0) x := x-1 { }

Strongest Postconditions

- Here are a number of valid Hoare Triples:
  - {x = 5} x := x * 2 { true }
  - {x = 5} x := x * 2 { x > 0 }
  - {x = 5} x := x * 2 { x = 10 || x = 5 }
  - {x = 5} x := x * 2 { x = 10 }
    - All are true, but this one is the most *useful*
    - x=10 is the *strongest postcondition*

- If {P} S {Q} and for all Q' such that {P} S {Q'}, Q $\Rightarrow$ Q', then Q is the strongest postcondition of S with respect to P
  - check: x = 10 $\Rightarrow$ true
  - check: x = 10 $\Rightarrow$ x > 0
  - check: x = 10 $\Rightarrow$ x = 10 || x = 5
  - check: x = 10 $\Rightarrow$ x = 10

Weakest Preconditions

- Here are a number of valid Hoare Triples:
  - {x = 5 && y = 10} z := x / y { z < 1 }
  - {x < y && y > 0} z := x / y { z < 1 }
  - {y ≠ 0 && x / y < 1} z := x / y { z < 1 }
    - All are true, but this one is the most *useful* because it allows us to invoke the program in the most general condition
    - y ≠ 0 && x / y < 1 is the *weakest precondition*

- If {P} S {Q} and for all P' such that {P'} S {Q}, P' ⇒ P, then P is the weakest precondition *wp*(S,Q) of S with respect to Q

Hoare Triples and Weakest Preconditions

- $\{P\}\ S\ \{Q\}$ holds if and only if $P \Rightarrow wp(S,Q)$
  - In other words, a Hoare Triple is still valid if the precondition is stronger than necessary, but not if it is too weak

- Question: Could we state a similar theorem for a strongest postcondition function?
  - e.g. $\{P\}\ S\ \{Q\}$ holds if and only if $sp(S,P) \Rightarrow Q$
  - A: Yes, but it's harder to compute

Quick Quiz

Consider the following Hoare triples:

A) y=1 { z = y + 1 } x := z * 2 { x = 4 }

B) y > 2 { y = 7 } x := y + 3 { x > 5 }

C) y!=0 { false } x := 2 / y { true }

D) { y < 16 } x := 2 / y { x < 8 }

- Which of the Hoare triples above are valid?

- Considering the valid Hoare triples, for which ones can you write a stronger postcondition? (Leave the precondition unchanged, and ensure the resulting triple is still valid)

- Considering the valid Hoare triples, for which ones can you write a weaker precondition? (Leave the postcondition unchanged, and ensure the resulting triple is still valid)

Hoare Logic Rules

- Assignment
  - { P } x := 3 { x+y > 0 }
  - What is the weakest precondition P?
    - What is most general value of y such that 3 + y > 0?
    - y > -3

Hoare Logic Rules

- Assignment
  - { P } x := 3 { x+y > 0 }
  - What is the weakest precondition P?
- Assignment rule
  - *wp*(x := E, P) = [E/x] P
    - Resulting triple: { [E/x] P } x := E { P }
  - [3 / x] (x + y > 0)
  - = (3) + y > 0
  - = y > -3

Hoare Logic Rules

- Assignment
  - $\{ P \}\ x := 3*y + z\ \{ x * y - z > 0 \}$
  - What is the weakest precondition P?
- Assignment rule
  - *wp*(x := E, P) = [E/x] P
  - $[3*y+z / x]\ (x * y - z > 0)$
  - $= (3*y+z) * y - z > 0$
  - $= 3*y^2 + z*y - z > 0$

Hoare Logic Rules

- Sequence
  - $\{ P \}\ x := x + 1;\ y := x + y\ \{ y > 5 \}$
  - What is the weakest precondition P?
- Sequence rule
  - $wp(S;T, Q) = wp(S, wp(T, Q))$
  - $wp(x{:=}x{+}1;\ y{:=}x{+}y,\ y{>}5)$
  - $= wp(x{:=}x{+}1,\ wp(y{:=}x{+}y,\ y{>}5))$
  - $= wp(x{:=}x{+}1,\ x{+}y{>}5)$
  - $= x{+}1{+}y{>}5$
  - $= x{+}y{>}4$

Hoare Logic Rules

- Conditional
  - { P } if x > 0 then y := z else y := -z { y > 5 }
  - What is the weakest precondition P?
- Conditional rule
  - *wp*(if B then S else T, Q)
    $= B \Rightarrow wp(S,Q) \ \&\& \ \neg B \Rightarrow wp(T,Q)$
  - *wp*(if x>0 then y:=z else y:=-z, y>5)
  - $= x>0 \Rightarrow wp(y{:=}z, y>5) \ \&\& \ x \leq 0 \Rightarrow wp(y{:=}-z, y>5)$
  - $= x>0 \Rightarrow z > 5 \ \&\& \ x \leq 0 \Rightarrow -z > 5$
  - $= x>0 \Rightarrow z > 5 \ \&\& \ x \leq 0 \Rightarrow z < -5$

Quick Quiz

Fill in the missing pre- or post-conditions with predicates that make each Hoare triple valid.

A) { x = y } x := y * 2 { }

B) { } x := x + 3 { x = z }

C) { } x := x + 1; y := y * x { y = 2 * z }

D) { } if (x > 0) then y := x else y := 0 { y > 0 }

Hoare Logic Rules

- Loops
  - { P } while (i < x) f=f*i; i := i + 1 { f = x! }
  - What is the weakest precondition P?

- Intuition
  - Must prove by induction
    - Only way to generalize across number of times loop executes
  - Need to guess induction hypothesis
    - Base case: precondition P
    - Inductive case: should be preserved by executing loop body

Proving loops correct

- First consider *partial correctness*
  - The loop may not terminate, but if it does, the postcondition will hold
- {P} while B do S {Q}
  - Find an invariant Inv such that:
    - $P \Rightarrow Inv$
      - The invariant is initially true
    - { Inv && B } S {Inv}
      - Each execution of the loop preserves the invariant
    - $(Inv \ \&\& \ \neg B) \Rightarrow Q$
      - The invariant and the loop exit condition imply the postcondition

Loop Example

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

    j := j + 1;
    s := s + a[j];

end
{ s = (Σi | 0≤i<N • a[i]) }
```

How can we find a loop invariant?

Replace N with j
Add information on range of j

Loop Example

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;
{ 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
while (j < N) do
    {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
    j := j + 1;
    s := s + a[j];
    {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
end
{ s = (Σi | 0≤i<N • a[i]) }
```

Quick Quiz

Consider the following program:

```
{ N >= 0 }
i := 0;
while (i < N) do
    i := N
{ i = N }
```

Which of the following loop invariants are correct? For those that are incorrect, explain why.

A) i = 0

B) i = N

C) N >= 0

D) i <= N

Proof Obligations

- Invariant is initially true

  $\{ N \geq 0 \}$
  $j := 0;$
  $s := 0;$
  $\{ 0 \leq j \leq N \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i]) \}$

- Invariant is maintained

  $\{0 \leq j \leq N \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i]) \ \&\& \ j < N\}$
  $j := j + 1;$
  $s := s + a[j];$
  $\{0 \leq j \leq N \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i]) \}$

- Invariant and exit condition implies postcondition

  $0 \leq j \leq N \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i]) \ \&\& \ j \geq N$
  $\quad \Rightarrow s = (\Sigma i \mid 0 \leq i < N \bullet a[i])$

Proof Obligations

- Invariant is initially true

  { N ≥ 0 }
  { 0 ≤ **0** ≤ N && 0 = (Σi | 0≤i<**0** • a[i]) } *// by assignment rule*
  j := 0;
  { 0 ≤ j ≤ N && **0** = (Σi | 0≤i<j • a[i]) } *// by assignment rule*
  s := 0;
  { 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }

- Need to show that:

  (N ≥ 0) ⇒ (0 ≤ 0 ≤ N && 0 = (Σi | 0≤i<0 • a[i]))
= (N ≥ 0) ⇒ (0 ≤ N && 0 = **0**) *// 0 ≤ 0 is true, empty sum is 0*
= (N ≥ 0) ⇒ (0 ≤ N) *// 0=0 is true, P && true is P*
= **true**

Proof Obligations

- Invariant is maintained

  $\{0 \le j \le N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i]) \;\&\&\; j < N\}$

  $\{0 \le \mathbf{j+1} \le N \;\&\&\; s+a[\mathbf{j+1}] = (\Sigma i \mid 0 \le i < \mathbf{j+1} \bullet a[i])\}$ *// by assignment rule*

  $j := j + 1;$

  $\{0 \le j \le N \;\&\&\; \mathbf{s+a[j]} = (\Sigma i \mid 0 \le i < j \bullet a[i])\}$ *// by assignment rule*

  $s := s + a[j];$

  $\{0 \le j \le N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i])\}$

- Need to show that:

  $(0 \le j \le N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i]) \;\&\&\; j < N)$

  $\quad \Rightarrow (0 \le j+1 \le N \;\&\&\; s+a[j+1] = (\Sigma i \mid 0 \le i < j+1 \bullet a[i]))$

= $(0 \le j < N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i]))$

  $\quad \Rightarrow (\mathbf{-1 \le j < N} \;\&\&\; s+a[j+1] = (\Sigma i \mid 0 \le i < j+1 \bullet a[i]))$ *// simplify bounds of j*

= $(0 \le j < N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i]))$

  $\quad \Rightarrow (-1 \le j < N \;\&\&\; s+a[j+1] = (\Sigma i \mid 0 \le i < j \bullet a[i]) + \mathbf{a[j]})$ *// separate last element*

// we have a problem – we need a[j+1] and a[j] to cancel out

Where's the error?

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

    j := j + 1;
    s := s + a[j];

end
{ s = (Σi | 0≤i<N • a[i]) }
```

Need to add element ***before*** incrementing j

Corrected Code

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

    s := s + a[j];
    j := j + 1;

end
{ s = (Σi | 0≤i<N • a[i]) }
```

Proof Obligations

- Invariant is maintained

{0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
{0 ≤ j +1 ≤ N && **s+a[j]** = (Σi | 0≤i<j+1 • a[i]) }   *// by assignment rule*
s := s + a[j];
{0 ≤ **j +1** ≤ N && s = (Σi | 0≤i<**j+1** • a[i]) }   *// by assignment rule*
j := j + 1;
{0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }

- Need to show that:

(0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N)
    ⇒ (0 ≤ j +1 ≤ N && s+a[j] = (Σi | 0≤i<j+1 • a[i]))
= (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
    ⇒ (**-1 ≤ j < N** && s+a[j] = (Σi | 0≤i<j+1 • a[i])) *// simplify bounds of j*
= (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
    ⇒ (-1 ≤ j < N && s+a[j] = (Σi | 0≤i<j • a[i]) **+ a[j]**) *// separate last part of sum*
= (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
    ⇒ (-1 ≤ j < N && s = (Σi | 0≤i<j • a[i]))   *// subtract a[j] from both sides*
= **true**   *// 0 ≤ j ⇒ -1 ≤ j*

Proof Obligations

- Invariant and exit condition implies postcondition

$0 \leq j \leq N \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i]) \ \&\& \ j \geq N$

$\quad \Rightarrow s = (\Sigma i \mid 0 \leq i < N \bullet a[i])$

$= \quad 0 \leq j \ \&\& \ \mathbf{j = N} \ \&\& \ s = (\Sigma i \mid 0 \leq i < j \bullet a[i])$

$\quad \Rightarrow s = (\Sigma i \mid 0 \leq i < N \bullet a[i])$

// because $(j \leq N \ \&\& \ j \geq N) = (j = N)$

$= \quad 0 \leq \mathbf{N} \ \&\& \ s = (\Sigma i \mid 0 \leq i < \mathbf{N} \bullet a[i]) \Rightarrow s = (\Sigma i \mid 0 \leq i < N \bullet a[i])$

// by substituting N for j, since j = N

= **true** *// because P && Q ⇒ Q*

Quick Quiz

- For the program below and the invariant i <= N, write the proof obligations. The form of your answer should be three mathematical implications.

```
{ N >= 0 }
{ 0 <= N }
i := 0;
{ i <= N }
while (i < N) do
   { i <= N && I < N}
   { N <= N }
   i := N
   { i <= N }
{ i <= N && i >= N }
{ i = N }
```

- Invariant is initially true:

- Invariant is preserved by the loop body:

- Invariant and exit condition imply postcondition:

Invariant Intuition

- For code without loops, we are simulating execution directly
  - We prove one Hoare Triple for each statement, and each statement is executed once
- For code with loops, we are doing *one* proof of correctness for *multiple* loop iterations
  - Don’t know how many iterations there will be
  - Need our proof to cover all of them
  - The invariant expresses a *general* condition that is true for every execution, but is still strong enough to give us the postcondition we need
  - This tension between generality and precision can make coming up with loop invariants hard

Session Summary

- While testing can find bugs, formal verification can assure their absence
- Hoare Logic is a mechanical approach for verifying software
  - Creativity is required in finding loop invariants, however

Further Reading

- C.A.R. Hoare. **An Axiomatic Basis for Computer Programming.** *Communications of the ACM* 12(10):576-580, October 1969.