COMP 150-SEN
Software Engineering Foundations

Program Verification

Spring 2019

(Based on lecture notes by Hal Perkins, CS 211, Cornell)
Software Specifications

• A specification defines the behavior of an abstraction

• It is a contract between the user and provider
  - Provider’s code must implement the specification
  - Providers can change the implementation as long as it still meets the specification
  - Users that depend on the implementation could be in trouble
    - Should only rely on the specification
It’s Hard to Write Good Specs

• Very difficult to get developers to write specs
  ▪ Even harder to keep them up to date

• Having specs in a separate document from the code almost guarantees failure
  ▪ Rational for javadoc and similar: extract a standalone specification from the code and embedded comments

• Hard to accurately and formally capture all properties of interest
  ▪ Always finding important details not specified
Specifications Are Useful

• There are lots of subtle algorithms and data structures
  - Internal specs/invariants vital to correct implementation
  - Example: Binary search tree
    - All nodes reachable from left child have smaller key than current node
    - All nodes reachable from right child have larger key than current node

• In the real world, much coding effort goes into modifying previously written code
  - Often originally written by somebody else
  - Documenting and respecting specs avoids a mess
Formal vs. Informal Specifications

- Informal
  - If the array $d$ is sorted, and some element of $d$ is equal to $x$, then $\text{find}(d, x)$ returns the index of $x$ …

- Formal
  - $(\forall i \text{ such that } 0<i<d.\text{length } . \ d[i-1] \leq d[i] \text{ and} \n  \quad (\exists j \text{ such that } 0\leq j<d.\text{length } . \ d[j] = x)$
  - $\Rightarrow \text{find}(d,x) = j$ …

```java
static int find(int[] d, int x) { ... }
```
Pros and Cons

• **Formal specs**
  - Force you to be very clear
  - Might be checkable with automated tools
    - Either at compile time (static checking) or run time (dynamic checking)

• **Informal specs**
  - Some important properties are hard to express formally
    - Sometimes just difficult and long
    - Sometimes don’t have the necessary formal notation
  - Some people are intimidated by formal specs
Type of External Specifications

• Specifications on method
  - Preconditions: What must be true before call
  - Postconditions: What is made true after call
    - Often relates return values to argument values
  
```
// precondition: d is sorted
// postcondition:
// (ret≥0 ∧ d[ret] == x) ∨ (ret=-1 ∧ x∉d)
static int find(int[] d, int x) { ... }
```

• Potentially also: side effects
  - What heap state is modified by calling the method
• Potentially also: performance
  - Exact bounds? Maybe high-level: random access fast, insertion slow…
Types of Internal Specifications

- Specs appearing within the code itself
- Loop invariants: condition that must hold at the beginning of each iteration of a loop

```
static int find(int[] d, int x) {
    // inv: ∀0≤j<i . d[j]≠x
    for (int i=0; i<d.length; i++) {
        if (d[i]==x) return i;
    }
}
```

- More on these shortly!

- Object invariants (more later!)

```
class Buffer {
    char[] bytes;
    int filled; // inv: 0≤filled<bytes.length
}
```
Specification and Subtyping

• The Liskov Substitution Principle:

  $S$ is a subtype of $T$ if anyone expecting a $T$ can be given an $S$ instead

• How do specs of original and overridden method relate?

```java
// pre: d is sorted
// post: (ret≥0 ∧ d[ret] == x) ∨ (ret=-1 ∧ x∉d)
static int find(int[] d, int x) { ... }
```

• If we override `find`, can the new method

  ▪ Have true as a precondition?
  ▪ Have `d is sorted` and $∃i. d[i]=x$ as a precondition?
  ▪ Have postcond $(ret=-1 ∧ x∉d)$ (only)?
  ▪ Throw `NoSuchElementException` rather than returning -1 if $x∉d$
General Rule for Overriding

- Postconditions must be related as follows
  - We want: Anyone who expects an A can be given a B (subtyping definition)
  - So, if someone is expecting postA to hold after the call, it’s okay if postB holds as long as postB ⇒ postA
• Preconditions must be related as follows
  
  - We want: Anyone who expects an A can be given a B (subtyping definition)
  - So, if someone makes \( \text{preA} \) hold before the call, it’s okay to call a method expecting \( \text{preB} \) as long as \( \text{preA} \Rightarrow \text{preB} \)
General Rule for Overriding (cont’d)

- Cumulatively, for $B \subset A$ ($B$ subtype of $A$) need
  - $\text{postB} \Rightarrow \text{postA}$
  - $\text{preA} \Rightarrow \text{preB}$

- Notice the direction flips between pre/post
  - Postcondition is covariant, precondition is contravariant
Javadoc

- Documentation as source code comments
  - Running `javadoc` on code creates separate html doc files

```java
/** Javadoc Comment for this class */
public class MyClass {
    /** Javadoc Comment for field text */
    String text;

    /** Given a sorted array, returns the index into the array of the given element, otherwise returns -1. */
    @param d array to search in, assumed sorted
    @param x the element to search for
    @returns i >= 0 when d[i] == x, and -1 when x does not occur in d */
    public static int find(int d[], int x) {
    }
}
```
Class MyClass

java.lang.Object
    MyClass

public class MyClass
extends java.lang.Object

Javadoc Comment for this class
    ...

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int</td>
<td>find(int[] d, int x)</td>
</tr>
<tr>
<td></td>
<td>Given a sorted array, returns the index into the array of the given element, otherwise returns -1.</td>
</tr>
</tbody>
</table>
Method Detail

find

public static int find(int[] d,
        int x)

Given a sorted array, returns the index into the array of the given element, otherwise returns -1.

Parameters:
d – array to search in, assumed sorted
x – the element to search for
Testing vs. Verification

“Program testing can be used to show the presence of bugs, but never to show their absence!” — Edsger Dijkstra, *Notes on Structured Programming*, 1970

- Testing only
- So then, how do we show the absence of bugs?

• Enter *program verification*
  - Given program and spec, *prove* the program meets the spec
  - Idea pioneered by
  - Difficult to make progress on idea for a long time, but there have been many recent successes
    - Warning: those successes require work far beyond what we’ll see in class
Predicates and Assertions

• We will talk about programs using *predicates*
  - A predicate is any boolean expression
  - Examples:
    ```
    true
    false
    x = 5    // not assignment!
    x > 5 ∨ x < 10
    y ≠ null
    ```

• An *assertion* is a predicate written in `{ }’s
  ```
  x = 4;
  y = x + 2;
  { x = 4 ∧ y = 6 }
  ```
  - `{ p }` indicates that p is true at that program point
Hoare Triples

- Putting an assertion before and after a statement creates a *Hoare Triple*

\[
\{Q\} \ S \ \{R\}
\]

- This means
  - If \(Q\) holds before executing \(S\) and
  - if \(S\) is executed and
  - if \(S\) terminates then
  - \(R\) will be true in the final state

- This is a *partial correctness* spec
  - It also holds if \(S\) doesn’t terminate

- \(Q\) is the precondition, \(R\) is the postcondition
Hoare Triple Examples

- If \( x \) is less than 0 and we increment \( x \), then \( x \) is less than 1
  - This Hoare triple is valid, i.e., it is true
    \[
    \{x<0\} \ x++; \ {x<1}
    \]

- If \( x \) is 4 and we set \( x \) to 5, then \( x \) is 5
  - This Hoare triple is valid valid
  - But notice the precondition is stronger than we need
    - Here, stronger means more restrictive
  - The weakest or most general possible precondition is
    \[
    \{x=4\} \ x=5; \ {x=5}
    \]

- Convention: use capital letters to refer to arbitrary values of variables in preconditions
  \[
  \{x=X\} \ y=x; \ {x=X \land y=X}
  \]
A Longer Example

- We can interleave assertions between statements to give us several triples that combine to show a sequence of statements correct

```java
{ true }
if (x ≤ y)
    { x ≤ y, hence x = min(x,y) }
    z = x;
    { z = min(x,y) }
else
    { y < x, hence y = min(x,y) }
    z = y;
    { z = min(x,y) }
{ z = min(x,y) }
```

- `{}`'s omitted for if/else to avoid confusion with triples
Two Annotated Loops

- Both of these are valid
  - But notice the one on the right might not terminate!
    - If $x < 0$

\[
\begin{align*}
\{ & x \geq 0 \} \\
\text{while } (x \neq 0) & \ x--; \\
\{ & x = 0 \}
\end{align*}
\]

\[
\begin{align*}
\{ & \text{true} \} \\
\text{while } (x \neq 0) & \ x--; \\
\{ & x = 0 \}
\end{align*}
\]
Total vs. Partial Correctness

• We can also define a total correctness version of Hoare triples

\[
[Q] \ S \ [R]
\]

• Meaning
  - If \( Q \) holds before executing \( S \) and
  - if \( S \) is executed then
  - \( S \) terminates and
  - \( R \) will be true in the final state

• Total and partial correctness are only different for loops
Two Annotated Loops, Again

- Now the version on the left is valid, but the version on the right is invalid
  - Because the code on the right might not terminate
Proving Hoare Triples Correct

• We have examples of valid and invalid triples
  - We’ve been figuring this out informally so far
  - We need to establish ground rules for when triples are valid

• We need to develop a logic
  - In particular, we want a mechanical system of proof rules
    - We will stipulate that the proof rules are correct
    - Thus, we will say a triple is valid if and only if we can show that it is valid according to the proof rules

• Same rules work for partial and total correctness
  - Except for termination, which we’ll discuss separately
Proof Rule: Sequencing

If \{A\} S1 \{B\} and \{B\} S2 \{C\} are valid
Then \{A\} S1; S2 \{C\} is valid

• Example
  - The following two triples are valid
    \[
    \{ \text{true} \} \ x = 0; \ { x=0 } \\
    \{ x=0 \} \ y = x + 1; \ { x=0 \land y=1} 
    \]
  - Thus, the following is valid
    \[
    \{ \text{true} \} \ x = 0; \ y = x + 1; \ { x=0 \land y=1} 
    \]
Proof Rule: Conditional

If \( \{A \land b\} \ S_1 \ \{B\} \) and \( \{A \land \neg b\} \ S_2 \ \{B\} \) are valid
Then \( \{A\} \ \text{if } b \ \text{then } S_1 \ \text{else } S_2 \ \{B\} \) is valid

- **Example**
  - To show the following is valid
    \[
    \{x=X\} \ \text{if } (x<0) \ \text{then } y = -x; \ \text{else } y = x; \ {y=|X|}
    \]
  - We need to show the following two triples are valid, which they are
    \[
    \{x=X \land x<0\} \ y = -x; \ {y=|X|}
    \]
    \[
    \{x=X \land x\geq0\} \ y = x; \ {y=|X|}
    \]
Assignment and Substitution

• The proof rule for assignment has the form

```
{“fact about e”} x=e; {“fact about x”}
```

- For example:

```
{“2*n is even”} x=2*n; {“x is even”}
```

• To make this precise, we need to define substitution

- We write $P[x\mapsto e]$ to mean predicate $P$ in which variable $x$ is replaced by expression $e$

- $(x=y)[x\mapsto w] = (w=y)$
- $(x=y)[x\mapsto x+2] = (x+2=y)$
- $(y=x+x)[x\mapsto z+x+y] = (y=z+x+y+z+x+y)$
- $(z=x*y)[x\mapsto a+b] = (z=x+(a*b))$ // added ()'s
Proof Rule: Assignment

\{A[x\mapsto e]\} \ x=e \ \{A\} \ is \ valid

- • Examples:

\{ \ \text{true} \ \} \ x = 2; \ \{ \ x=2 \ \}

\{ \ x+1\geq 0 \ \} \ x = x+1; \ \{ \ x\geq 0 \ \}

- • or, simplifying:

\{ \ x\geq -1 \ \} \ x = x+1; \ \{ \ x\geq 0 \ \}

- • Notice this rule is very easy to apply “backwards”

\{ \ ??? \ \} \ x = x-y; \ \{ \ x*y + y*y = 5 \ \}

- • What is ??

- • Must be \( (x-y)*y + y*y = 5 \)

- • Or \( x*y - y*y + y*y = 5 \) \ or \( x*y = 5 \)
Proof Rule: Consequence

• Sometimes we need to add some implications to get the assertions we really want

If \( \{A\} \ S \ \{B\} \) is valid and \( A' \Rightarrow A \) and \( B \Rightarrow B' \)
Then \( \{A'\} \ S \ \{B'\} \) is valid

• Example: the assignment rule gives us

\[
\{ x>0 \} \quad y = x + 1; \quad \{ y>0 \}
\]

- Since \( x=0 \Rightarrow x>-1 \), we can also conclude

\[
\{ x=0 \} \quad y = x + 1; \quad \{ y>0 \}
\]

- Also, since \( y>0 \Rightarrow y>-5 \), we can also conclude

\[
\{ x=0 \} \quad y = x + 1; \quad \{ y>-5 \}
\]
Checking Validity Mechanically

• Given an assignment and a postcondition, we can use the assignment rule to compute the *weakest precondition*
  - I.e., least restrictive precondition that will cause the postcondition to hold

• This suggests a mechanical way to check whether \{A\} S \{B\} holds for straight line code
  - Start with \(B\)
  - Apply the assignment rule repeatedly to work back to the start, yielding the weakest precondition \(A'\)
  - Check whether \(A \Rightarrow A'\)

• We can generalize this to other constructs
Weakest Preconditions

- Define $wp(S, B)$ to be the weakest precondition of $B$, as follows
  - $wp(x=e, B) = B[x\mapsto e]$
  - $wp(S_1; S_2, B) = wp(S_1, wp(S_2, B))$
  - $wp(if\ E\ then\ S_1\ else\ S_2, B) = E\Rightarrow wp(S_1, B) \land \neg E\Rightarrow wp(S_2, B)$
Assignment and Pointers

• Warning: The assignment rule does not yield the weakest precondition in the presence of pointers

\[
\{ \ *y = 5 \ \} \ *x = 5; \ \{ \ *x + \ *y = 10 \ \}
\]

- The above is true, but the actual weakest precondition is

\[
\{ \ *y = 5 \ \text{or} \ x = y \} \ *x = 5; \ \{ \ *x + \ *y = 10 \ \}
\]

• To extend this style or reasoning to include pointers, see work on separation logic
Loops

• There is no mechanical way to automatically check whether loops are correct

• Two key problems
  ▪ Weakest preconditions might not terminate, because it needs to keep going backward through the loop repeatedly
  ▪ We’d like to also know whether loops terminate, which weakest preconditions doesn’t reason about

• Instead, we’re going to develop a pen-and-paper way of reasoning about loops
  ▪ You’ll see how this is translated into practice in a future lecture
Summation in a Loop

• Goal: prove the following triple correct

\[
\text{true} \\
\text{sum} = a[0] ; \\
k = 0 ; \\
\text{while} \ (k \neq n-1) \{ \\
\quad k = k + 1 ; \\
\quad \text{sum} = \text{sum} + a[k] ; \\
\} \\
\text{sum} = a[0] + a[1] + \ldots + a[n-1] \\
\]

- Notice this is total correctness, i.e., we need to prove the loop terminates
- (We will ignore concerns about the array length)
Loop Invariants

- Key idea: think about one loop iteration, generically
- A loop invariant is an assertion that holds at the beginning and end of each execution of the loop

\[
[Q] \text{S_init; while}(B) \{ S \} [R]
\]
- Let \( I \) be the loop invariant

- To prove this loop correct
  1. Show \([Q] \text{S_init} [I]\) (invariant holds before loop)
  2. Show \([I \land B] S [I]\) (invariant holds across loop)
  3. Show \(I \land \neg B \Rightarrow R\) (post holds after loop)
Loop Invariant Example

1. \[ [ \text{true} ] \]
   \[ \text{sum} = a[0]; \]
   \[ k = 0; \]
   \[ \text{while } (k \neq n-1) \{ \]
     \[ k = k + 1; \]
     \[ \text{sum} = \text{sum} + a[k]; \]
   \[ \} \]
   \[ [ \text{sum} = a[0]+a[1]+\ldots+a[n-1] ] \]

I: \((\text{sum} = a[0]+\ldots+a[k] \land 0 \leq k < n)\)

- Holds by assignment rules
Loop Invariant Example (cont’d)

2. \[[\text{sum}=a[0]+...+a[k] \land 0\leq k<n \land k \neq n-1]\]
   \[
   k=k+1; \quad \text{sum} = \text{sum} + a[k];
   \]

\[[\text{sum}=a[0]+...+a[k] \land 0\leq k<n]\]
   - Holds by assignment rules and consequence
Loop Invariant Example (cont’d)

I: \((\text{sum} = a[0]+...+a[k] \land 0 \leq k < n)\)

3. \((\text{sum} = a[0]+...+a[k] \land 0 \leq k < n \land k = n-1) \Rightarrow \)

\[
\text{sum} = a[0]+a[1]+...+a[n-1]
\]

- Holds by standard logical reasoning
Bound Function for Termination

• A bound function $t$ is
  - An integer-valued expression defined in terms of program variables
  - $t$ must strictly decrease with every loop iteration
  - $t \geq 0$ always, so that when it reaches 0 the loop terminates

• I.e., a bound function is an upper bound on the number of remaining loop iterations
Bound Function Example

Check that \( t \) is a bound function

- \( t \) strictly decreases with each step because \( k \) increases by 1
- \( t \) can never go below zero because the loop terminates with \( k=n-1 \)

Therefore, the loop terminates
Invariants are Abstractions

• An *invariant* is just an assertion at a program point

• We’ve seen four kinds of invariants
  ▪ Precondition: invariant at method entry
  ▪ Postconditions: invariant at method exit
  ▪ Object invariant: invariant about fields that holds at beginning and end of every method within a class
  ▪ Loop invariant: invariant at every iteration of a loop

• Invariants create an abstraction barrier
  ▪ Invariant must be established by code before it
  ▪ Code after it can rely on the invariant being true

• Invariants are a powerful tool for understand code!
  ▪ Try to think about what always holds at a program point
  ▪ Add assertions to code to confirm your understanding