Formal Verification

Basic Verification Strategy

compare behavior to intent

System

Model of system behavior

Verifier

intent

results
Intent

- Usually, originates with requirements, refined through design and implementation

- Formalized by specifications
  - Often expressed as formulas in mathematical logic

- Different types of intent
  - E.g., performance, functional behavior
  - Each captured with different types of formalisms

- Specification of behavior/functionality
  - What functions does the software compute?
  - Often expressed using predicate logic

Compare behavior to intent

- Can be done informally—by manual inspections
  - Code Inspections

- Can be done selectively
  - Checking assertions during execution

- Can be done formally
  - With theorem proving
    - Usually with automated support
    - Called Proof of Correctness or Formal Verification
      - Proof of “correctness” is dangerously misleading
  - With static analysis for restricted classes of specifications
Theorem Proving based Verification

- **Behavior** inferred from semantically rich program model
  - generally requires most of the semantics of the programming language
  - employs symbolic execution

- **Intent** captured by predicate calculus specifications (or another mathematically formal notation)

Theorem-Proving based Verification Strategy

System \[\xrightarrow{\text{Inferred using symbolic execution}}\] Model of system behavior \[\xrightarrow{}\] Theorem prover

Predicate logic assertions

Intent

Results
Floyd Method of Inductive Assertions

- Show that given input assertions, after executing the program, program satisfies output assertions
  - Break the program into program fragments, and prove each fragment separately
    - Divide and conquer approach
    - Use induction to prove that fragments with loops behave as intended
    - By transitivity, proofs of all the fragments imply that the program is correct
- Show that the program must terminate

Creating Program Fragments

- Place assertions at the start, final, and intermediate points in the code
- Any path is composed of sequences of program fragments that start with an assertion, are followed by some assertion free code, and end with an assertion
  - \( A_s, C_1, A_2, C_2, A_3, \ldots A_{n-1}, C_{n-1}, A_f \)
- Show that for every executable path, if \( A_s \) is assumed true and the code is executed, then \( A_f \) is true
Pictorially: A single path (noloops)

Must show:
assume $A_i$, then execute Code $C_i$, necessarily $\Rightarrow A_{i+1}$

use symbolic execution (i.e., forward substitution)
Why does this work?
suppose P is an arbitrary path through the program
can denote it by
P = A_0 C_1 A_1 C_2 A_2 ... C_n A_n
where
A_0 - Initial assertion
A_n - Final assertion
A_i - Intermediate assertions
C_i - Loop free, uninterrupted, straight-line code

If it has been shown that
\forall i, 1 \leq i < n: A_i C_i \Rightarrow A_{i+1}
Then, by transitivity
A_0 \Rightarrow A_n

Obvious problem

- How do we do this for all paths?
  - Infinite number of paths
    - Must find a way to deal with loops
    - Use mathematical induction
Mathematical Induction (Digression)

- Goal: prove that a given property holds for all elements of a collection
- Approach:
  - show property holds for "first" element
  - show that if property holds for element i, then it must also hold for element i + 1
- Often used when direct analytic techniques are too hard or complex

Example: How many edges in Cn?

- Theorem:
- Let Cn = (Vn, En) be a complete, unordered graph on n nodes,
  then |En| = n * (n-1)/2
Example: How many edges in $C_n$?

- To show that this property holds for the entire set of complete graphs, $\{C_i\}$, by induction:
  1. show the property is true for $C_1$
  2. show if the property is true for $C_n$, then the property is true for $C_{n+1}$

Example: How many edges in $C_n$?

- show the property is true for $C_1$:
  - graph has 1 node, 0 edges
  - $|E_1| = n(n-1)/2 = 1(0)/2 = 0$
Example: How many edges in $C_n$?

- Assume true for $C_n$: $|E_n| = \frac{n(n-1)}{2}$
- Graph $C_{n+1}$ has one more node, but $n$ more edges (one from the new node to each of the $n$ old nodes)
- Thus, want to show $|E_{n+1}| = |E_n| + n = \frac{(n+1)n}{2}$

Proof: $|E_{n+1}| = |E_n| + n = \frac{n(n-1)}{2} + n$

by substitution

\[
= \frac{n(n-1)}{2} + \frac{2n}{2}
\]

by rewriting

\[
= \frac{n(n-1) + 2n}{2}
\]

by simplification

\[
= \frac{n(n-1+2)}{2}
\]

by simplification

\[
= \frac{n(n+1)}{2}
\]

by rewriting

End Digression

How to handle loops -- unroll them

```
input assertion
n do_while predicate1
  n+1 if predicate2
  n+2 then code;
  n+3 else code;
  n+4 end;
  n+5 output assertion;
```

loop invariant

```
input assertion
n
n+1
n+2
n+3
n+4
n+5
output assertion
```

```
output assertion
n+1
n+2
n+3
n+4
n+5
output assertion
```

```
output assertion
n+1
n+2
n+3
n+4
n+5
output assertion
```

```
output assertion
n+1
n+2
n+3
n+4
n+5
output assertion
```
Better -- find loop invariant \((A_i)\)

subpaths to consider:
- \(C_1\): Initial assertion \(A_0\) to final assertion \(A_f\)
- \(C_2\): Initial assertion \(A_0\) to \(A_i\)
- \(C_3\): \(A_i\) to \(A_i\)
- \(C_4\): \(A_i\) to final assertion \(A_f\)

Similar to an inductive proof

Consider all paths through a loop

subpaths to consider:
- \(C_1\): \(A_0\) to \(A_f\)
- \(C_2\): \(A_0\) to \(A_i\)
- \(C_3\): \(A_i\), false branch, \(A_i\)
- \(C_4\): \(A_i\), true branch, \(A_i\)
- \(C_5\): \(A_i\), false branch, \(A_f\)
- \(C_6\): \(A_i\), true branch, \(A_f\)
Assertions

- Specification that is intended to be true at a given site in the program
- Use three types of assertions:
  - initial: sited before the initial statement
    e.g., precondition
  - final: sited after the final statement
    e.g., postcondition
  - intermediate: sited at various internal program locations
    subject to the rule:
    - a "loop invariant" is true on every iteration thru the loop

Floyd’s Inductive Verification Method (more carefully stated)

- Specify initial and final assertions to capture intent
- Place intermediate assertions so as to "cut" every program loop
- For each pair of assertions where there is at least one executable (assertion-free) path from the first to the second, create and prove the verification condition:
  - assume that the first assertion is true
  - show that for all (assertion-free, executable) paths from the first assertion to the second, that the second assertion is true
- This above establishes "partial correctness"
- Show that the program terminates
  This establishes "total correctness"
Example

Assume we have a method, called FindValue, that takes as input three parameters: a table that is an array of values where the index starts at zero, n is the current number of values in table (with entries from 0 to n-1), and a key that is also of type value. FindValue returns the smallest index of the element in table that is equal to the value of key. If no element of table is equal to key, then a new last element with that value is added to the table and that index is returned.

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

// postconditions
ensures (table[result] == key)
ensures \forall(int i=0; i < \result; i++)
(table[i] != key)
ensures \result>=0 & & \result<n
ensures \result==n => table.size()>=n+1
ensures \result<n => table.size() >=n

Example: FindValue

Int FindValue (int table[ ], int n, int key) {
    boolean found;
    found=false;
    current = 0;
    while (not found && current < n) {
        if (table[current] == key)
            found = true;
        else
            current = current + 1;
    }
    if (not found) {
        table[current] = key;
    }
    return (current);
}
FindValue

found = false;
current = 0;

not found &&
current < n

T

<table[current]> := key

T

table[current] = key

F

current =
current + 1;

F

found = true

T

return(current)

What needs to be done

- Must define all the intermediate assertions
- Must create all the verification conditions
- Must prove each verification condition
- Must prove termination
found = false;  
current = 0;

not found && current < n

T

not found

table[current] = key

F

found = true

current = current + 1;

not found

return(current)

T

T

F

F

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n <= table.size();

// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++)
\( (table[i] != key) \)
ensures current >= 0 && current <= n
ensures current == n => table.size() >= n+1
ensures current < n => table.size() >= n

Where should we put intermediate assertions?

Where should we put intermediate assertions?

Where should we put intermediate assertions?
found = false; current = 0;
not found && current < n

found = true

T
F

T
F

T
F

T
F

T
F

What do we need to prove?:
precondition (code1) invariant
precondition (code2) loop invariant
loop invariant (code3) loop invariant: 2 cases?
loop invariant (code4) invariant: 2 cases?
invariant (code 5) postconditions: 2 cases

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n <= table.size();

// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++)
    (table[i] != key)
ensures current>=0 && current<=n
ensures current==n => table.size()>=n+1
ensures current<n => table.size()>=n

What do we need to prove?:
precondition (code1) invariant
precondition (code2) loop invariant
loop invariant (code3) loop invariant
loop invariant (code4) invariant
invariant (code 5) postconditions: 2 cases

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n = table.size();

// verification conditions

What should the loop invariant be?

- Ensures $\text{found} = \text{false}$
- Ensures $\text{current} \geq 0$ and $\text{current} < n$
- Ensures $\forall (\text{int } i = 0; i < \text{current}; i++) (\text{table}[i] \neq \text{key})$

The loop invariant is:

$\text{found} = \text{false}$
$\text{current} = 0$
$\text{not found and current} < n$

The postconditions are:

- Ensures $\text{table}[\text{current}] = \text{key}$
- Ensures $\forall (\text{int } i = 0; i < \text{current}; i++) (\text{table}[i] \neq \text{key})$
- Ensures $\text{current} \geq 0$ and $\text{current} < n$
- Ensures $\text{found} = \text{false}$
- Ensures $\text{current} \geq 0$ and $\text{current} < n$
- Ensures $\forall (\text{int } i = 0; i < \text{current}; i++) (\text{table}[i] \neq \text{key})$

What do we need to prove?

- Precondition $\{\text{code1}\}$ invariant $\{\text{code1}\}$
- Precondition $\{\text{code2}\}$ loop invariant $\{\text{code2}\}$
- Loop invariant $\{\text{code3}\}$ loop invariant $\{\text{code3}\}$
- Loop invariant $\{\text{code4}\}$ invariant $\{\text{code4}\}$
- Invariant $\{\text{code5}\}$ postconditions: 2 cases
Proving one verification condition

**Precondition (code2) Loop invariant**

\[ \text{requires } n \geq 0; \]
\[ \text{requires } \text{key} \neq \text{null}; \]
\[ \text{requires } \text{table} \neq \text{null}; \]
\[ \text{requires } n \leq \text{table.size()}; \]

\{\text{found} = \text{false} \\
\text{current} = 0 \\
\text{not found} \& \& \text{current } < n\} \\
\text{ensures } (\text{found} = \text{false}) \\
\text{ensures } \text{current} > 0 \& \& \text{current } < n \\
\text{ensures } \forall(i = 0; i < \text{current}; i++) \quad (\text{table}[i] = \text{key}) \]

**Proof: Precondition (code2) Loop invariant**

Executing

\{\text{found} = \text{false} \\
\text{current} = 0 \\
\text{not found} \& \& \text{current } < n\}

\(a\) By execution found = false

\(b\) By execution, current = 0 and current < n

\implies n > current = 0

\therefore current > 0 \& \& current < n

\(c\) int i=0; i < 0 is empty, so stmt

\(\text{(table}[i] = \text{key})\) is true

---

**Diagram**

- **Preconditions**
  - \(\text{found} = \text{false}\)
  - \(\text{current} = 0\)
- **Invariant**
  - \(\text{not found} \& \& \text{current } < n\)
  - \(\text{table}[\text{current}] = \text{key}\)
- **Loop invariant**
  - \(\text{found} = \text{true}\)
  - \(\text{current} = \text{current} + 1\)
- **Postconditions**
  - \(\text{ensures } (\text{found} = \text{false})\)
  - \(\text{ensures } \text{current} > 0 \& \& \text{current } < n\)
  - \(\text{ensures } \forall(i = 0; i < \text{current}; i++) \quad (\text{table}[i] = \text{key})\)

**Cases**

- **Precondition (code1) Invariant**
- **Precondition (code2) Loop invariant**
- **Loop invariant (code3) Loop invariant**
- **Loop invariant (code4) Loop invariant**
- **Invariant (code 5) Postconditions: 2 cases**

*Need to represent initial versus final values*
loop invariant \{code3\} loop invariant

ensures (found == false)
ensures current >= 0 \&\& current < n
ensures \forall (int i=0; i < current; i++)
  (table[i] != key)

\{ not found \&\& current' < n 
If table[current'] == key then
  found = true
else current = current' + 1
not found \&\& current < n \}
ensures (found == false)
ensures current >= 0 \&\& current' < n
ensures \forall (int i=0; i < current; i++)
  (table[i] != key)

2 cases to consider: true and false branch

For the true case, only found needs to distinguish initial from changed values

a) True case: using symbolic execution
Given found' == false \&\& current < n
By execution (found = true)
By substitution (not found = false) => infeasible path
Thus, only have to consider the false branch

loop invariant \{code3\} loop invariant

ensures (found == false)
ensures current >= 0 \&\& current < n
ensures \forall (int i=0; i < current'; i++)
  (table[i] != key)

\{ not found \&\& current < n 
If table[current'] == key then
  found = true
else current = current' + 1
not found \&\& current < n \}
ensures (found == false)
ensures current >= 0 \&\& current' < n
ensures \forall (int i=0; i < current; i++)
  (table[i] != key)

2 cases to consider: true and false branch

For the false case, only current needs to distinguish initial from changed values
Proving another verification condition

loop invariant {code3} loop invariant
ensures (found == false)
ensures current' >= 0 && current' < n
ensures \( \forall (int i=0; i < current'; i++): table[i] != key \)

Proof:

a) By execution, not found = true =>
   found == false

b) By execution, (current < n)
   Given current > 0 and by execution that
   current = current' + 1
   => current = current' + 1 >= 0 + 1
   => current >= 0
   therefore current >= 0 && current < n

c) Given by (a) above, found = false so
   by execution
   table[current'] := key
   current = current' + 1
   By given, forall(int i=0; i < current'; i++):
   (table[i] := key)
   => forall(int i=0; i < current'+1; i++):
   (table[i] := key)
   => forall(int i=0; i < current; i++):
   (table[i] := key)

What remains to be done?

- Must prove all the verification conditions
  - precondition {code1} invariant
  - precondition {code2} loop invariant
  - loop invariant {code3} loop invariant
  - loop invariant {code4} invariant
  - invariant {code 5} postconditions: 2 cases
- Must prove termination
Must Prove Termination

```java
Int FindValue (int table[], int n, int key) {
    boolean found;
    found=false;
    current = 0;
    while (not found && current < n) {
        if (table[current] == key)
            found = true;
        else
            current = current + 1;
    }
    if (not found) {
        table[current] = key;
    }
    return (current);
}
```

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

// postconditions
ensures (table[\result] == key)
ensures ∀(int i=0; i < \result; i++)
   (table[i] != key)
ensures \result>=0 &\ result<=n
ensures \result=n ⇒ table.size()>=n+1
ensures \result<n ⇒ table.size()>=n

After proof is done, FindValue is correct

```java
Int FindValue (int table[], int n, int key) {
    boolean found;
    found=false;
    current = 0;
    while (not found && current < n) {
        if (table[current] == key)
            found = true;
        else
            current = current + 1;
    }
    if (not found) {
        table[current] = key;
    }
    return (current);
}
```

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

// postconditions
ensures (table[\result] == key)
ensures ∀(int i=0; i < \result; i++)
   (table[i] != key)
ensures \result>=0 &\ result<=n
ensures \result=n ⇒ table.size()>=n+1
ensures \result<n ⇒ table.size()>=n
After proof is done, FindValue is correct

```c
Int FindValue (int table[], int n, int key) {
    boolean found;
    found=false;
    current = 0;
    while (not found && current < n) {
        if (table[current] == key)
            found = true;
        else
            current = current + 1;
    }
    if (not found) {
        table[current] = key;
    }
    return (current);
}

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

// postconditions
ensures (table[result] == key)
ensures \forall(int i=0;i < result; i++)
    (table[i] != key && table[i] = table[i]')
ensures result>=0 & result<=n
ensures result==n => table.size()>=n+1
ensures result<n => table.size() >=n
ensures \forall(int i=n+1;i < table.size(); i++)
    (table[i] = table[i]')
ensures n=n'
ensures key=key'
```

So beware:

- Formal verification can be used to prove that:
  Given the input conditions, after executing the program, the output condition is satisfied
- It does not prove that the program is correct because:
  - The postcondition may be incorrect or incomplete
  - The precondition might be wrong
  - The invariants might be wrong
  - The proof might be incorrect
Example: FindValue version 2

Boolean FindValue (int table[], int n, int key) {
    current = 0;
    while (table[current] != key && current < n) {
        current = current + 1;
    }
    if (current = n) {
        table[current] = key;
    }
    return (current);
}

// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n <= table.size();

// postconditions
ensures (table[result] == key)
ensures \forall(int i=0;i < result; i++)
    (table[i] != key & table[i] = table[i])
ensures result >= 0 & result <= n
ensures result == n => table.size() >= n+1
ensures result + n => table.size() > n
ensures \forall(int i=n+1;i < table.size(); i++)
    (table[i] = table[i])
ensures n = n'
ensures key = key'

Floyd’s Inductive Verification Method

Specify initial and final assertions to capture intent
Place intermediate assertions so as to "cut" every program loop
For each pair of assertions where there is at least one executable (assertion-free) path from the first to the second, create and prove the verification condition:
   assume that the first assertion is true
   show that for all (assertion-free, executable) paths from the first assertion to the second, that the second assertion is true
This above establishes “partial correctness”
Show that the program terminates
   This establishes “total correctness"