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 human eye
  - Cleanroom
  - 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 properties
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

Model of system behavior

Theorem prover

intent

predicate logic assertions

inferred using symbolic execution

results
**Floyd Method of Inductive Assertions**

- Show that given *input assertions*, after executing the program, program satisfies *output assertions*
  - show that each program fragment behaves as intended
  - use induction to prove that all fragments, including loops, behave as intended
- show that the program must terminate
Mathematical Induction

- **goal**: prove that a given property holds for all elements of a set

- **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 $C_n$

Theorem:

let $C_n = (V_n, E_n)$ be a complete, unordered graph on $n$ nodes,

then $|E_n| = n \times (n-1)/2$
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 one node, 0 edges

\[ |E_1| = \frac{n(n-1)}{2} = \frac{1(0)}{2} = 0 \]
Example: How many edges in $C_n$

assume true for $C_n$: $|E_n| = n(n-1)/2$

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 $|E_{n+1}| = |E_n| + n = (n+1)(n)/2$

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

by substitution

$= n(n-1)/2 + 2n/2$

by rewriting

$= (n(n-1)+2n)/2$

by simplification

$= (n(n-1+2))/2$

by simplification

$= n(n+1)/2$

by simplification

$= (n+1)(n)/2$

by rewriting
Floyd's Method of inductive verification (informal description)

- 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

initial assertion

intermediate assertions

final assertion

STRAIGHT-LINE CODE

$A_i \rightarrow C_i \rightarrow A_{i+1}$
Must be sure:
assuming $A_i$,
then executing Code $C_i$,
necessarily $\Rightarrow A_i + 1$

by forward substitution
$\Rightarrow$ symbolic execution
**Why does this work?**

Suppose \( P \) is an arbitrary path through the program can denote it by

\[
P = A_0 \ C_1 \ A_1 \ C_2 \ A_2 \ldots \ C_n \ A_n
\]

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

\[
A_0 \ldots \Rightarrow A_n
\]
Obvious problems

• How do we do this for a path?
• How do we do this for all paths?
  • Infinite number of paths
    • Must find a way to deal with loops
How to handle loops -- unroll them

\[ \text{input assertion} \]

\[
n \quad \text{do}\_\text{while predicate1} \\
n+1 \quad \text{if predicate2} \\
n+2 \quad \text{then code ;} \\
n+3 \quad \text{else code ;} \\
n+4 \quad \text{end;} \\
n+5 \quad \text{output assertion ;} \\
\]
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 \(A_f\)

Basically an inductive proof
Consider all paths through a loop

subpaths to consider:

C₁: A₀ to A₉
C₂: A₀ to A₁
C₃: A₁, false branch, A₁
C₄: A₁, true branch, A₁
C₅: A₁, false branch, A₉
C₆: A₁, true branch, A₉
**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
  - **final**: sited after the final statement
  - **intermediate**: sited at various internal program locations

  subject to the rule:
  - every loop iteration shall pass through the site of at least one intermediate assertion
  - 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,
  • 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 establishes “partial correctness”
• Show that the program terminates
  • This establishes “total correctness”
Floyd-Hoare axiomatic proof method

assertions are preconditions and postconditions on some statement or sequence of statements

\[ P \{S\} Q \]

if P is true before S is executed and S is executed then Q is true

P is the precondition;
Q is the postcondition

Also written \{P\} S \{Q\}
Floyd-Hoare axiomatic proof method

- as in Floyd's inductive assertion method, we construct a sequence of assertions, each of which can be inferred from previously proved assertions and the rules and axioms about the statements and operations of the program
- to prove $P(S)Q$, we need some axioms and rules about the programming language
Hoare axioms and proof rules

take a simple programming language that deals only with integers and has the following types of constructs:

• assignment statement  
  \( x := f \)

• composition of a sequence of statements  
  \( S_1, S_2 \)

• conditional (alternative statements)  
  if \( B \) then \( S_1 \) else \( S_2 \)

• iteration  
  while \( B \) do \( S \)
Axioms and proof rules

• axiom of assignment
  \[ P \{ x:=f \} Q, \]
  where \( Q \) is obtained from \( P \) by substituting \( f \) for all occurrences of \( x \) in \( P \) (symbolic execution)

• rule of composition
  \[ P \{ S_1, S_2 \} Q \Rightarrow \exists \ P_1, \ P(S_1)P_1 \land P_1(S_2)Q \]
  Using Hoare's notation, this is written as

\[
\begin{align*}
P(S_1)P_1, \ P_1(S_2)Q \\
P \{ S_1, S_2 \} \ Q
\end{align*}
\]
Proof Rules (continued)

• rule for the alternative statement
  \[ P\{\text{if } B \text{ then } S_1 \text{ else } S_2 \} Q \Rightarrow P\{B \land S_1\} Q \land P\{\neg B \land S_2\} Q \]

• Hoare's notation

\[ \frac{P\{B \land S_1\} Q, \ P\{\neg B \land S_2\} Q}{P\{\text{if } B \text{ then } S_1 \text{ else } S_2 \} Q} \]
Proof Rules (continued)

rule of iteration

\[ P \{ \text{while } B \text{ do } S \} Q \Rightarrow P\{\neg B\}Q \land \exists I \exists P \{ B \land S \} I \land I\{ B \land S \} I \land I\{ \neg B \} Q \]

\[ P\{\neg B\}Q, P \{ B \land S \} I, I\{ B \land S \} I, I\{ \neg B \} Q \]

\[ P \{ \text{while } B \text{ do } S \} Q \]
weakest precondition

- in Hoare technique $P\{S\}Q$

$S1$:
read $x, y$
$z := y$
while $x > 0$ do
  $z := z + 1$
  $x := x - 1$
endwhile;

$S2$:
read $x, y$
$z := x + y$

suppose $P = \{x \geq 0\}$
$Q = \{z = x + y\}$
- then we can prove $P\{S1\}Q$ and $P\{S2\}Q$, but we can also prove $true\{S2\}Q$
- $S2$ is provable for any $x, y$, but $S1$ is provable only for $x \geq 0$
Dijkstra's Axiomatic Semantics

- In general, there are many correct pre- and post-conditions for a given program
- Seek the strongest post condition and the weakest precondition
  - $P \Rightarrow P'$; $P$ is **stronger** than $P'$ and $P'$ is **weaker** than $P$
**Rules of consequence**

- If $P \Rightarrow P'$ and $Q' \Rightarrow Q$ and $P'\{S\}Q'$ then $P\{S\}Q$

\[
\begin{align*}
\text{P}\{S\}Q' , \ Q' \Rightarrow Q \\
\hline
\text{P}\{S\}Q \\
\text{P}\Rightarrow P' , \ P'\{S\}Q \\
\hline
\text{P}\{S\}Q \\
\text{P}\Rightarrow P' , \ P'\{S\}Q' , \ Q' \Rightarrow Q \\
\hline
\text{P}\{S\}Q
\end{align*}
\]
Formal Verification Process

• determine input, output and loop invariant assertions
• identify all paths between two assertions (with no intervening assertions) and form the corresponding verification condition or lemma
• prove each verification condition (partial correctness)
• prove that the program terminates