Formal Verification

Prof. Leon Osterweil
Computer Science 520/620
Spring 2013

Some Examples of “Relations”

• Executing this code must compute this function
• This code must conform to that design element
• This compiled code came from this compiler
• This design element addresses those requirements
• These lower level requirements are elaborations of these higher level requirements
• This is the date by which that test must be passed
• Component invocations conform to component abstract interface specifications
• Documentation describes the actual system
• ETC.....

Some Examples of “Relations”

• Executing this code must compute this function
• This code must conform to that design element
• This compiled code came from this compiler
• This design element addresses those requirements
• These lower level requirements are elaborations of these higher level requirements
• This is the date by which that test must be passed
• Component invocations conform to component abstract interface specifications
• Documentation describes the actual system
• ETC.....

Evaluation of Static Analysis

• Strengths:
  – Can demonstrate the absence of faults
  – Proofs can be automatically generated and proven
  – Algorithms are fast (low-order polynomial)
  – No need to generate test data
  – You know when you are done
• Weaknesses
  – Behavior specification is a model with inaccuracies
    » Not all paths are executable
  – Only certain classes of faults analyzable
    » Mostly sequence specific
    » Weak on functionality
Symbolic Execution

- Specification of Intent: Formulae, functions
- Specification of Behavior: Functions derived from annotated flowgraph, symbol table
  - Annotate nodes with function(s) computed there
  - Specify path to be studied
  - Compute function(s) computed as composition(s) of functions at path nodes, constraints of path edges
  - Comparison: Solving simultaneous constraints; symbolic algebra
- Results: Demonstrations that given paths computed the right function(s)
- What to do about the fact that most programs have many (perhaps an unlimited amount of) paths?

Formal Verification

- Prove that a program must always deliver the specified functional behavior
  - Regardless of what path is executed
  - Even if there are unlimited numbers of paths
- Requires a formal specification of the desired functional behavior.
- Requires a formal specification of the computation that is to deliver this behavior
- Requires solid mathematical reasoning

Previously this was misleadingly called “Proof of Correctness”

Formal Verification

- Intent
  - Usually specification of functionality
  - What function(s) does the software compute?
  - Sometimes accuracy, timing, ...
- Behavior
  - Inferred from semantically rich program model
  - Generally requires most of semantics of programming language
  - Generally uses symbolic execution
- Comparison
  - Use of formal mathematics (e.g. predicate logic)
Example Assertion

```
X := Y;
Time := Y * 2.0 * T;
ASSERT Time > 0.0;
<rest of code>
```

Use of Assertions in Testing

- Zoom in on internal workings of the program
- Examine behaviors at internal program locations while the program is executing
  - Augments examining only final outputs
- Assertions: Specifications of intended relations among the values of program variables
  - Development of increasingly elaborate assertion languages
  - Often: Checking relations between code and design
- Comparison: Runtime evaluation of assertions
  - Facilities for programming reactions to violations
  - Previously seen as a debugging aid
  - Now we will see it as a key support for static verification

Use of Assertions in Verification

- Assertion: Specification of a condition that is intended to be true at a specific given site in the program text
- Floyd's Method assertions are written in Predicate Logic
- In Floyd's Method there are three types of assertions
  - Initial, $A_i$: Sited at the program initial statement
  - Final, $A_f$: Sited at the program final statement
  - Intermediate $A_i$: Often called a "loop invariants"
- Sited at various program locations subject to the rule:
  
  EVERY LOOP ITERATION (CFG CYCLE) SHALL PASS THRU THE SITE OF AT LEAST ONE INTERMEDIATE ASSERTION

Net Effect: Every program execution sequence is divided into a finite number of segments of non-looping code bounded on each end by a predicate logic assertion

Mathematical Induction

- Goal: prove that a given property holds for all elements of a set
- Approach:
  - (initial step) show property holds for "first" element
  - (induction step) 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 $C_n = (V, E)$ be a complete, unordered graph on $n$ nodes,

then $|E| = n \cdot (n-1)/2$

```
Example Assertion

```

```
Initial Step

- show the property is true for $C_1$:
  - graph has one node, 0 edges
  - $|E_1| = n(n-1)/2 = 1(0)/2 = 0$

Induction Step

- 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)
  - so, $|E_{n+1}| = n(n-1)/2 + n$
    - $n(n-1)/2 + 2n/2 = n(n-1) + 2n/2$
    - $= n(n+1)/2$
    - $= (n+1)(n+1)/2$
    - $= (n+1)(n)/2$

Floyd’s Method of inductive assertions (informal description)

- Place assertions at the start, final, and intermediate points in the code.
- Any path is composed of sequences of program fragments all of which:
  - start with an assertion
  - are followed by some code that is:
    - Assertion-free
    - Loop-free
  - and end with an assertion
  - eg. ($A_i =$) $A_0$, $C_1$, $A_1$, $C_2$, $A_2$, ..., $A_n$, $C_n$$, A_n$, $C_o$, $A_o$(=A)
- Show that for any executable path, if $A_i$ is assumed true for any $A_i$ and code $C_i$ is executed, then $A_{i+1}$ must always be true

Induction in Floyd’s Method

- Initially
  - Verify that the Initial Assertion is correct
- Induction on length of execution path
  - Prove that function computed by all execution paths of length $L + 1$ are correct
    - Provided that all execution paths of length $L$ compute the correct function
- Inductive step is the hard one (as usual)
  - Proof relies on the fact that there are only a (relatively) small number of path segments in any program.

Pictorially

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

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

by forward substitution

Why does this work?
suppose $P =$ arbitrary path through the program
can denote it by

$P = A_s C_1 A_1 C_2 A_2 ... C_n A_f$

where

- $A_s =$ Initial assertion
- $A_f =$ 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 induction
$A_s .... \Rightarrow A_f$

AGAIN: There are only a small number of these in any realistic program

Loop Invariants (Loop Breakers)
• Problem: infinite number of paths
  – Must find a way to deal with loops
• Solution: Assertion, $A_i$, that is
  – True for any number of loop iterations
  – "connects up" to adjacent assertions
• Such an assertion:
  – Is invariant with respect to loop iterations
  – Must be embedded in (break) every loop

A loop invariant must capture the essence
Of the work that the loop is to perform

Schematic Example of a Loop Invariant
$A_i$ is a loop invariant because of its relation to other assertions:

NOTE THAT:
$A_s$ false branch, $\Rightarrow A_i$
$A_s$ true branch, $\Rightarrow A_i$

BUT ALSO:
Initial assertion $A_s$ to $A_i$ $\Rightarrow A_i$
$A_s$ false branch, $\Rightarrow$ final assertion $A_i$
$A_s$ true branch, $\Rightarrow$ final assertion $A_i$

How Many Path Segments?

$A_s \rightarrow A_i$
$A_i$ true $\rightarrow A_i$
$A_i$ false $\rightarrow A_i$
$A_i$ true $\rightarrow A_i$
$A_i$ false $\rightarrow A_i$
$A_i \rightarrow A_i$

Floyd’s Method (carefully stated)
• Specify initial, final assertions to capture intent
• Intermediate assertions "cut" every program loop
• For each pair of assertions with an 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"
Sort Example

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;

ASSERT INNER
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

ASSERT OUTER

values[i] ≥ values[j] for all j, where i < j ≤ size

ASSERT INNER

values[i] ≥ values[k] for all i < k ≤ j
**Assert Initial**

size > 0

and

size < 1000

**Sort Example**

```plaintext
procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

assert initial
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;
end do;

assert inner
end do;

assert outer
end do;

assert final
end sort;
```

**Assert Final**

For all $i < j \leq size$:

$\text{values}[i] \geq \text{values}[j]$

And: $\text{values'}$ is a permutation of $\text{values}$:

For all $i < j \leq size$, there exists a $k$ such that $\text{values}[k] = \text{IN}_{i}$

**Lemma 1: Assert Inner, false => Assert Inner**

```plaintext
procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

assert initial
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;
end do;

assert inner
end do;

assert outer
end do;

assert final
end sort;
```
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL
  do for i = 1 to size-1
    do for j = i+1 to size
      if values[j] > values[i] then
        temp := values[i];
        values[i] := values[j];
        values[j] := temp;
      end if
    end do;
  end do;

ASSERT INNER
  values[i] ≥ values[k] for all i < k ≤ j-1
  Symbolic Execution:
  Symb. Vals.              Path Conditions
  values[j] = IN
  values[j] > values[i]

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;

Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;

Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;

Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;

Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;

Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT OUTER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT INNER:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

ASSERT FINAL:
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
      end if;
  end do;
end do;

end sort;
Lemma 2: Assert Inner, true => Assert Inner

```
procedure sort(values, size);
    assert values real array[0:0], temp real, i, j, size integer;
    assert initial:
    do for i = 1 to size-1
        do for j = i+1 to size
            if values[j] > values[i] then
                temp := values[j];
                values[j] := values[i];
                values[i] := temp;
            end do;
        end do;
    end do;
    end sort;
```

Loop Invariants

- Should be able to identify loop invariant for every loop that you write
- Should be able to state initial and final assertions too
- Maybe not in formal mathematics
- Should be able to reason about how they support overall program logic

An Example: Wensley’s Algorithm

```
procedure Wensley (P: input, Q: input, E: input, Y: output);
    declare P, Q, E, A, B, D real;
    assert input P, Q, E, 0 < E;
    A := 0.0; B := Q / 2.0; D := 1.0; Y := 0.0;
    do while (D >= E)
        if (P - A - B >= 0.0) then {Y := Y + (D / 2.0); A := A + B};
        B := B / 2.0; D := D / 2.0;
        end do;
    end Wensley;
```

What does Wensley's algorithm do?

- Approximating P/Q (= Y) with error ≤ E
- On the kth iteration of the loop
  \[ A_k = a_1 Q^2 + a_2 Q^2 + \ldots + a_k Q^2 \]
  \[ B_k = Q^k \]
  \[ D_k = 2^k \]
  \[ Y_k = a_1 2+ a_2 2^2 + \ldots + a_k 2^k \]
  \[ a_i \in \{0,1\} \]

What does Wensley’s algorithm do?

- Since 0 ≤ P/Q ≤ 1, then P/Q can be estimated as a sum of the series
  \[ a_1 2 + a_2 2^2 + \ldots + a_k 2^k \]
  \[ a_i \in \{0,1\} \]
- Therefore
  - \( Y_k \) is the computed value of the quotient
  - Given \( Y_k \), \( A_k \) is the computed dividend P
  - \( D_k \) is the computed divisor
  - \( P \cdot (A_k + B_k) \) says when to add \( 2^{(k+1)} \) to \( Y_{k+1} \) (\( a_{k+1} = 1 \))
  \[ B_k = 2^{(k+1)} \cdot Q \]
Assertions

Initial:
\[ A_0: \{(0 \leq P < Q) \land (0 < E)\} \]

Final:
\[ A_i: \{(P/Q - E) < Y \leq (P/Q)\} \]

Intermediate:
\[ A_i: \{(A = Q \cdot Y) \land (B = Q \cdot (D/2)) \land ((P/Q) - D) < Y \leq (P/Q)\} \]

\( Y \) is within the computed error \( D \) of \( P/Q \).

Summary of Four Lemmas Needed

I. Initial assertion, \( A_0 \), to \( A_i \)
II. \( A_i \), false branch, to \( A_i \)
III. \( A_i \), true branch, to \( A_i \)
IV. \( A_i \), to \( A_i \), final assertion

Lemma I: \( A_s \) to \( A_i \)

\( A_s: \) Initial Assertion
\[ (0 \leq P < Q) \land (0 < E) \]

Input P, Q, E

\[ \begin{align*}
A &\leftarrow 0.0 \\
B &\leftarrow Q/2 \\
D &\leftarrow 1.0 \\
Y &\leftarrow 0.0
\end{align*} \]

\( A \rightarrow A_i \)

Proof:
1) \( A = 0; \ Q \cdot Y = Q \cdot 0 = 0 \)
2) \( B = Q/2 = Q \cdot 1/2 = Q \cdot D/2 \)
3) \( D = 1 = 2^0 \)
4) \( P/Q - 1 < Y = 0 \leq P/Q \)

because \( 0 \leq P/Q < 1 \) given by input assertion

Lemma II: \( A_i \), false branch, \( A_i \)

\[ \begin{align*}
A &\leftarrow Q \cdot Y \\
B &\leftarrow Q \cdot (D/2) \\
D &\leq P/Q - 1 < Y = 0 \leq P/Q
\end{align*} \]

\( A \rightarrow A_i \)

• Need to establish that \( A_i \) is a correct relation among variable values after loop execution, based on assumption that \( A_i \) was correct among variable values before loop execution

• Notation:
  - \( A, B, D, Y \) are original values of variables
  - \( A', B', D', Y' \) are values after loop execution

• Symbolic execution gives:
  - \( A' = A; B' = B/2; D' = D/2; Y' = Y \)

\[ \begin{align*}
A &\leftarrow A \\
P &\leftarrow Q \cdot B \\
A &\leftarrow A \\
P &\leftarrow Q \cdot B \\
A &\leftarrow A \\
P &\leftarrow Q \cdot B \\
A &\leftarrow A \\
P &\leftarrow Q \cdot B
\end{align*} \]

Proof of lemma II
### Proof of Lemma II

**Step 1:**

\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \] for some integer \( k \)
\[ P/Q - D < Y \]
\[ \leq D = 2 \]
\[ A = Q \cdot Y \]

**Step 2:**

\[ A = A - B \]
\[ B = B/2 \]
\[ D = D/2 \]
\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \] for some integer \( k \)
\[ P/Q - D < Y \]
\[ \leq D = 2 \]
\[ A = Q \cdot Y \]

**Step 3:**

\[ A = A - B \]
\[ B = B/2 \]
\[ D = D/2 \]
\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \] for some integer \( k \)
\[ P/Q - D < Y \]
\[ \leq D = 2 \]
\[ A = Q \cdot Y \]

### Lemma III: True branch: \( A_i \)

- **Proof:**
  - From symbolic execution we know:
  - \( A' = A + B \)
  - \( B' = B/2 \)
  - \( D' = D/2 \)
  - \( Y' = Y + D/2 \)
  - \( A' = Q \cdot Y' \)
  - \( B' = Q \cdot D'/2 \)
  - \( D' = 2^{k-1} \) for some integer \( k \)
  - \( P/Q - D' < Y' \)
  - \( \leq P/Q \)

- **Lemma IV \( A_{i} \), \( A_{f} \)**

  - **False, \( A_{f} \)**

- **Lemma IV:**

  \[ A = (Q \cdot Y) \cdot (Q \cdot D/2) \] for some integer \( k \geq 0, k \in \mathbb{Z} \)
  \[ P/Q - D < Y \]

  **Proof:**

  - Given \((P/Q-D)<Y\) and \((D<E)\) \[ (P/Q-E)<(P/Q-D)<Y \]
  \[ (P/Q-E)<(P/Q-D)<Y \]

- **This is only partial correctness**

  - Must also prove termination
    - In general, cannot prove termination
    - For most programs, can usually do it by showing that each loop must terminate

  - For example: given that \((E>0)\), observe that \(D\) is halved on each iteration and \(E\) does not change, eventually \(D=E\) and the loop terminates.
Observations

- Proofs are long, tedious & often hard
- Assertions are hard to get right
- Invariants are difficult to get right.
  - need to be invariant, but also need to support overall proof strategy
- Proofs themselves often require deep program insight
  - Often require axioms about the domain

Deeper Issues

- Unsuccessful proof attempt \( \leadsto \) ???
  - incorrect software
  - incorrect assertions
  - incorrect placement of assertions
  - inept prover
  - any combination (or all) of the above
- Although failed proofs often indicate which of the above is likely to be the problem (especially to an astute prover)

Deeper Issues

- Undecidability of Predicate calculus -- no way to be sure when you have a false theorem
  - There is no sure way to know when you should quit trying to prove a theorem (and change something)
- Proofs are generally much longer than the software being verified
  - Suggests that errors in the proof are more likely than errors in the software being verified

Mathematics as a "social process"

- Belief in a proof is a social process
  - Informally describe proof
  - Distribute an informal write-up to colleagues
  - Formal write-up is refereed
  - Accepted paper gets read by wider audience
  - Proof/Theorem is used
  - Increases confidence
- Despite this, mathematical proofs are often wrong

Specification Problem

- Real programs are not captured by simple mathematical algorithms
  - E.g. "This software correctly identifies faces"
  - Error processing issues
  - User interface issues
- Resulting specifications are
  - Large
  - Mathematically unappealing
  - Probably not complete
  - Hard to capture intent

Specification Problem

- Specification & program are not independent representations
  - Proof may lack complete axiomatic details
- Very labor intensive
  - Loop invariants - usually manual
  - Input and output assertions - manual
  - Verification conditions - can be automated
Software Tools Can Help

• Proof Checkers:
  – Scrutinize the steps of a proof and determine if they are sound
  – Identify the rule(s) of inference, axiom(s), etc. needed to justify each step
  – How to know if the proof checker is right (verify it? with what? ....)

Software Tools Can Help

• Verification Assistants
  – Facilitate precise expression of assertions
  – Accept rules of inference
  – Accept axioms
  – Construct statements of needed lemmas
  – Check proofs
  – Assist in construction of proofs (theorem provers)

Human/computer collaboration

• Most successful -- human/computer collaboration
  » Human architects the proof
  » Computer attempts the proof (generally by exhaustive search of space of possible axioms and inferences at each step)
  » Human intervention after computer has tried for a while

Current Status:

• Have verified some non-trivial programs or important parts of programs
  – e.g., protocol verification
  – TOKENEER
• Improved theorem provers
• Improved specification languages
• Verification and testing/analysis research now viewed more as a continuum
  
  testing--> finite state verification--> verifications

Summary

• Verification has had a very positive impact on software engineering
  – major argument for structured programming
    » Dijkstra’s "goto's considered harmful" letter
    » one-in one-out structures easier to reason about
  – major impetus for abstract data types
    » centralized all changes to a data structures
    » input/output assertions for all operations

Formal Development

• Start with assertions, develop software artifacts to fulfill them
• A top-down approach
• Very popular in Europe: A hard sell in the U.S.
• Need to prove lemmas in higher level software dictates the functional requirements (eg, input/output assertion) pairs of lower level software artifacts.
• Also suggests the use of libraries of reusable verified software artifacts for commonly needed utilities
• This is Component-based software development
Integration of Testing Analysis and Formal Methods

- **Testing**
  - Is dynamic in nature, entailing execution of the program
  - Requires skillful selection of test data to assure good exercising of the program
  - Can show program executing in usage environment
  - Can support arbitrarily detailed examination of virtually any program characteristics and behavior
  - Is generally not suitable for showing absence of faults

- **Analysis**
  - Is static, operating on abstract program representations
  - Supports definitive demonstration of absence of faults
  - Generally only for certain selected classes of faults

- **Formal Methods**
  - Most through, rigorous, mathematical
  - Apply primarily to checking functional characteristics
  - Most human and cost intensive

- The types of capabilities are complementary; suggests need for skillful integration

---

Definitive reasoning benefits from both static and dynamic analysis techniques

- Religious wars of the 70’s
- Need testing to validate the "ground truth"
- Need static analysis to evaluate more than just what can be examined with testing
- Testing and analysis techniques currently being developed to work together
  - Testing -> Bug -> property -> verification -> counter examples -> feasibility analysis -> test cases -> testing ...

---

No Need To Restrict this only to Code

- Much of this is applicable to non-code artifacts
- Payoffs from detecting faults is greater the earlier it takes place
- How to apply this to non-code?