Formal Verification

Prof. Leon Osterweil
Computer Science 520/620
Spring 2012

Relations and Analysis

• A software product consists of
  – A collection of (types of) artifacts
  – Related to each other by myriad Relations
• The relations are essentially desiderata
  – At least initially
• Before the product can be trusted, the relations need to be verified/confirmed
• That is the role of analysis
  – Does the software do what it is supposed to do?
  – What are its capabilities and its strengths?
  – What is the nature of the artifact(s) that have been built?
  – What can I count on?
  – What should I worry about?
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)
Symbolic Execution

Specification of Intended Behavior
Function to be Computed

Specification of Actual Behavior
Formula Inferred from Actual Code

Functional Equivalence Theorem Prover
Comparison of Behavior to Intent

Proofs of Functional Correctness

Representing Computation

- **Symbolic names** represent the input values
- the **path value PV** of a variable for a path describes the value of that variable in terms of those symbolic names
- the **computation** of the path C[P] is described by the path values of the outputs for the path
Representing Conditionals

- an interpreted branch condition or interpreted predicate is represented as an inequality or equality condition
- the path condition PC describes the domain of the path and is the conjunction of the interpreted branch conditions
- the domain of the path D[P] is the set of input values that satisfy the PC for the path

Example program

```
procedure Contrived is
X, Y, Z : integer;
1    read X, Y;
if X ≥ 3 then
  3    Z := X+Y;
else
  4    Z := 0;
endif;
if Y > 0 then
  6    Y := Y + 5;
endif;
if X - Y < 0 then
  8    write Z;
else
  7    write Y;
endif;
end Contrived;
```

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X←In₁</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y ← In₂</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>Z ← In₁+In₂</td>
<td>true ∧ In₁≥3 = In₁≥3</td>
</tr>
<tr>
<td>5,6</td>
<td>Y ← In₁+5 In₁≥3 ∧ In₁&gt;0</td>
<td></td>
</tr>
<tr>
<td>7,9</td>
<td>(In₁≥3 ∧ In₁&gt;0 ∧ In₁ - (In₁+5 ≥0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= (In₁≥3 ∧ In₁&gt;0 ∧ (In₁-In₁)≥5)</td>
<td></td>
</tr>
</tbody>
</table>

Note: “Inᵢ” means the ith value taken as input to the program
Presenting the results

<table>
<thead>
<tr>
<th>Statements</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure Contrived is</td>
<td>X, Y, Z : integer;</td>
<td>1</td>
</tr>
<tr>
<td>read X, Y; if X ≥ 3 then</td>
<td></td>
<td>2,3</td>
</tr>
<tr>
<td>Z := X×Y; else</td>
<td></td>
<td>5,6</td>
</tr>
<tr>
<td>if Y &gt; 0 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y := Y + 5; endif;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if X - Y &lt; 0 then</td>
<td></td>
<td>7,9</td>
</tr>
<tr>
<td>write Z; else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Y; endif end Contrived</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P = 1, 2, 3, 5, 6, 7, 9
D[P] = { (In₁, In₂) | In₁≥3 ∧ In₂>0 ∧ In₁-In₂≥5 }
C[P] = PV.Y = In₂ +5
( PV.Z = In₁+ In₂ )

Results (feasible path)

P = 1, 2, 3, 5, 6, 7, 9
D[P] = { (In₁, In₂) | In₁≥3 ∧ In₂>0 ∧ (In₁-In₂)≥5 }
C[P] = PV.Y = In₂ +5
Evaluating another path

procedure Contrived is
X, Y, Z : integer;
1 read X, Y;
2 if X ≥ 3 then
3    Z := X+Y;
else
4    Z := 0;
endif;
5 if Y > 0 then
6    Y := Y + 5;
endif;
7 if X - Y < 0 then
8    write Z;
else
9    write Y;
endif;
end Contrived;

Stmts  PV   PC
1       X ← In_1      true
        Y ← In_2
2,3     Z ← In_1 + In_2 true ∧ In_1 ≥ 3 = In_1 ≥ 3
5,7     In_1 ≥ 3 ∧ In_2 ≤ 0
7,8     In_1 ≥ 3 ∧ In_2 ≤ 0 ∧ In_1 - In_2 < 0

P = 1, 2, 3, 5, 7, 8
D[P] = { (In_1, In_2) | In_1 ≥ 3 ∧ In_2 ≤ 0 ∧ In_1 - In_2 < 0 }
infeasible path!
Results (infeasible path)

\[(\ln_1 - \ln_2) < 0\]

Applications of Symbolic Evaluation

- Symbolic Testing
  - examination of path domain and computation for detecting failures
  - especially useful for scientific applications
- Path and Test Data Selection
  - select paths to cover structure and determine feasibility of condition
  - select data to satisfy path condition or "revealing" condition
- Debugging
  - examine symbolic representation for faulty data manipulation
- Verification
  - prove consistency of specification assertions
  - inductive assertion method for proving correctness...
  
  » \{I\} S \{O\} ...
Proof of Correctness

- **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 (eg. predicate logic)
  - Probably source of misleading name: PROOF of correctness
    - Proof is probably OK
    - Correctness is dangerously misleading

---

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 (eg. predicate logic)
  - Probably source of misleading name: PROOF of correctness
    - Proof is probably OK
    - Correctness is dangerously misleading
Floyd Method of Inductive Assertions

• Show that each program fragment behaves as intended
• Use induction to prove that all sequences of executable fragments behave as intended
• Show that the program must terminate

Assertion-Based 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
• Also useful as a debugging aid
<code sequence>
X := Y;
Time := Y * 2.0 * T;
**ASSERT Time > 0.0:**
<rest of code>

if ~(Time > 0.0) Then
Assertion_violation_handler;
</code>

**Assertion-Based Dynamic Testing**

- Specification of Intended Behavior
- Specification of Actual Behavior
- Functional Behavior Assertions
- Intermediate Execution Results
- Runtime Assertion Checking

Comparison of Behavior to Intent

Reports on Internal Failures
Use of Assertions

- 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, As: Sited at the program initial statement
  - Final, AF: Sited at the program final statement
  - Intermediate Ai: 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
Formal Verification

- Specification of Intended Behavior
- Specification of Actual Behavior
- Final and Intermediate First-Order Logic Assertions
- Symbolic Execution of Path Segments
- First-Order Logic Theorems and Proofs
- Proof of the Absence of All Functional Faults

Comparison of Behavior to Intent

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

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

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

1. Assume true for \( C_n: |E_n| = n(n-1)/2 \)
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)
3. So, \( |E_{n+1}| = n(n-1)/2 + n \)
   - \( = n(n-1)/2 + 2n/2 = (n(n-1) + 2n)/2 \)
   - \( = (n-1+2)/2 = n(n+1)/2 \)
   - \( = (n+1)((n-1)/2) \)
   - \( = (n+1)(n)/2 \)

Floyd’s Method of inductive assertions
(informal description)

1. Place assertions at the start, final, and intermediate points in the code.
2. Any path is composed of sequences of program fragments all of which:
   - Start with an assertion
   - Are followed by some assertion free code
   - And end with an assertion
     eg. \( (A_s =) A_0, C_1, A_1, C_2, A_2, \ldots A_{n-1}, C_n, A_n, C_n, A_{n+1}(=A_f) \)
3. 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

```
        intermediate assertions
          
      initial assertion

A_i  C_i  A_{i+1}

STRAIGHT-LINE CODE

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

by forward substitution

\[
\begin{array}{c}
A_i \\
C_i \\
A_{i+1}
\end{array}
\]

STRAIGHT-LINE CODE

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

$P = A_s \ C_1 \ A_1 \ C_2 \ A_2 \ldots C_n \ A_F$

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

$A_s \ldots \Rightarrow A_f$
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:
$A_i$, false branch, $\Rightarrow A_i$
$A_i$, true branch, $\Rightarrow A_i$

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

An Example:
Wensley's Algorithm

Procedure Wensley (P: input, Q: input, E: input, Y: output);
  --assume 0 ≤ P < Q, 0 < E
Declare P, Q, E, Y, A, B, D real;
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;
Wensley’s Algorithm

Input P, Q, E

A ← 0.0
B ← Q/2
D ← 1.0
Y ← 0.0

D ≥ E

P-A-B ≥ 0.0

Y ← Y+(D/2.0)
A ← A+B

B ← B/2.0
D ← D/2.0

What does Wensley's algorithm do?

• Approximating P/Q (=Y) with error ≤ E
• On the kth iteration of the loop

A_k = a_1Q^{-1} + a_2Q^{-2} + ... + a_kQ^{-k}
   a_i \in \{0,1\}
B_k = Q^{-k}
D_k = 2^{-k}
Y_k = a_12^{-1} + a_22^{-2} + ... + a_k2^{-k}
   a_i \in \{0,1\}
What does Wensley's algorithm do?

- since \(0 \leq P/Q < 1\), then \(P/Q\) can be estimated as a sum of the series
  \[a_12^{-1} + a_22^{-2} + \ldots + a_k2^{-k}\]
  \(a_i \in \{0, 1\}\)
  
- 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 error
  - \(P-(A_k + B_k)\) says when to add \(2^{-(k+1)}\) to \(Y_{k+1}\) \((a_{k+1} = 1)\)

\[B_k = 2^{-(k+1)} \times Q\]

Assertions

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

Final: \(A_F: \{((P/Q - E) < Y \leq (P/Q))\}\)

Intermediate:

\(A_i: \{(A = Q \times Y) \land (B = Q \times (D/2)) \land (k \geq 0, k \text{ integer} \land D = 2^{-k}) \land ((P/Q) - D) < Y \leq (P/Q)\}\)

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 $AF$, final assertion

Lemmas called verification conditions

Lemma I: $A_0$ to $A_I$

$A_0$: Initial Assertion

$(0 \leq P < Q) \land (0 < E)$

Input P, Q, E

A←0;
B←Q/2;
D←1;
Y←0;

$\Rightarrow A_I$.

A = Q * Y
B = Q * D/2
D = $2^{-k}$, k = 0
P/Q - D < Y ≤ P/Q
Lemma I: A₀ to Aᵢ

A₀: Initial Assertion
(0 ≤ P < Q) ∧ (0 < E)

Input P, Q, E
A←0;
B←Q/2;
D←1;
Y←0;

⇒ Aᵢ:
A = Q * Y
B = Q * D/2
D = 2⁻ᵏ, k=0
P/Q - D < Y ≤ P/Q

Proof:
1) A = 0; Q * Y = Q * 0 = 0
   A = Q * Y
2) B = Q/2 = Q * 1/2 = Q * D/2
3) D = 1 = 2⁻₀
4) P/Q - 1 < Y = 0 ≤ P/Q

because 0 ≤ P/Q < 1 given by input assertion

Lemma II: Aᵢ, false branch, Aᵢ

Aᵢ:
A = Q * Y
B = Q * D/2
D = 2⁻ᵏ for some integer k
P/Q - D < Y ≤ P/Q

D ≥ E [constraint]
P - A - B < 0 [constraint]
B ← B/2
D ← D/2

⇒ Aᵢ:
A = Q * Y
B = Q * D/2
D = 2⁻^(k+1)
P/Q - D < Y ≤ P/Q

D ≥ E
⇒ B ← B/2.0
D ← D/2.0

P - A - B ≥ 0.0
Proof of Lemma II

- Need to establish that A₁ is a correct relation among variable values after loop execution, based on assumption that A₁ 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:
  - $A' = A$, $B' = B/2$; $D' = D/2$; $Y' = Y$.

(Symbolic execution shows $A' = A, B' = B/2; D' = D/2; Y' = Y$)

Proof:

1) $A' = A = Q \times Y = Q \times Y'$

2) $B' = B/2 = (Q \times D/2)/2$
   \[ = (Q \times 2D'/2)/2 = Q \times D' /2 \]

3) $D' = D/2 = 2^{-k} / 2 = 2^{-(k+1)}$

4) a) $P - A - B < 0$ (constraint)
   \[ P - Q \times Y - Q \times D/2 < 0 \]
   \[ P/Q - Y - D/2 < 0 \text{ (uses } Q > 0) \]
   \[ P/Q - D/2 < Y \]
   but $D' = D/2$, so $P/Q - D' < Y'$

b) $Y \leq P/Q \Rightarrow Y' \leq P/Q$
**Lemma III: A₁; True branch; A₁**

**Proof Lemma III**

- **A₁:** \( A = Q \cdot Y \)
- \( B = Q \cdot D/2 \)
- \( D = 2^k \) for some integer \( k \)
- \( P/Q - D < Y \leq P/Q \)

**Proof:**

1. \( A' = A + B = Q \cdot Y + Q \cdot (D/2) = Q \cdot (Y + D/2) \)
2. \( B' = B/2 = Q \cdot D/2/2 = Q \cdot D' /2 \)
3. \( D' = 2^{(k+1)} \) for some integer \( k \)

**From symbolic execution we know:**

- \( A'' = A + B; \ B'' = B/2; \)
- \( D'' = D/2; \ Y'' = Y + D/2; \)

**We also know:** \( P - A - B \geq 0 \) and \( D \geq E \)

- \( P/Q - D < Y \leq P/Q \)
Lemma IV $A_I, A_F$

- $A_I$, false, $A_F$

\[
D \geq E
\]

\[D < E \quad \text{[code]}\]

$\Rightarrow A_F: ((P/Q-E) < Y \leq (P/Q))$

Proof:
Given $(P/Q-D)<Y\leq(P/Q)$ and $(D < E) \Rightarrow ((P/Q)-E)<((P/Q)-D)<Y\leq(P/Q) \Rightarrow A_F$
This is only partial correctness

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

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

```
Input P, Q, E
A ← 0.0
B ← Q/2
D ← 1.0
Y ← 0.0

P-A-B ≥ 0.0
D ≥ E

B ← B/2.0
D ← D/2.0

Y ← Y+(D/2.0)
A ← A+B
```
Deeper Issues

- Unsuccessful proof attempt ⇒ ???
  - 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 not 'mathematically' sound
- 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
- Improved theorem provers
- Improved specification languages
- Verification and testing/analysis research now viewed more as a continuum
  
  testing$\rightarrow$ finite state verification$\rightarrow$ 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 (e.g., 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 thorough, 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 ...

Testing & Analysis Process Architecture
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?

DEVELOPMENT PHASES

- Requirements Specification
- Architecting
- Implementation Designing
- Coding
- System Testing
- Software Sys. Test Plan
- Integration Test Plan
- Unit Test Plan
- System Test Plan
- Software Sys Testing
- Integration Testing
- Unit Testing

TEST PLANING

TESTING PHASES