Formal Verification

Prof. Leon Osterweil
Computer Science 520/620
Spring 2014

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

• 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.....
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)

Formal Verification

Specification of Intended Behavior

First-Order Logic

Theorems and Proofs

Proof of the Absence of All Functional Faults

Comparison of Behavior to Intent

Formal Verification

Specification of Actual Behavior

Symbolic Execution of Path Segments

Final and Intermediate First-Order Logic Assertions

Comparison of Behavior to Intent

Not necessarily code segments

First-Order Logic

Theorems and Proofs

Proof of the Absence of All Functional Faults

Example Assertion

\[ \text{X := Y;} \]
\[ \text{Time := Y \ast 2.0 \ast T;} \]
\[ \text{ASSERT Time > 0.0;} \]
\[ \text{<rest of code>} \]

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
Use of Assertions in Testing

- Assertions: Specifications of intended relations among the values of program variables
- Comparison: Runtime evaluation of assertions
  - Facilities for programming reactions to violations
- Examine behaviors at internal program locations while the program is executing
  - Augments examining only final outputs
- Previously seen as a debugging aid
- Now we will see it as a key support for static verification

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

Inserted by Tester

Automatically processed into

if ~(Time > 0.0) Then
Assertion_violation_handler;

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_s: 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_n, E_n) be a complete, unordered graph on n nodes,
then IE_n = n * (n-1)/2

Initial Step

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

**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. $(A_0, C_1, A_1, C_2, A_2, \ldots, A_{n-1}, C_{n-1}, A_n, C_n, A_{n+1} = A_f)$
- 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**

- Straight-line code

**Must be sure:**

assuming $A_i$, then executing Code $C_i$,

necessarily $\Rightarrow A_{i+1}$

by forward substitution

$A_i$  
$C_i$  
$A_{i+1}$

**Must be sure:**

assuming $A_i$, then executing Code $C_i$,

necessarily $\Rightarrow A_{i+1}$

by forward substitution

$A_i$  
$C_i$  
$A_{i+1}$

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

$$A_s \ldots \Rightarrow A_f$$

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:
  - $A_i$, 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 $A_f$

How Many Path Segments?

- $A_s \rightarrow A_i$
- $A_i$, true branch, $\rightarrow A_f$
- $A_i$, false branch, $\rightarrow A_f$
- $A_s \rightarrow A_f$

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"

A Little-JIL Definition of Floyd’s Method of Inductive Assertions
Rework Steps—Implemented by Recursion

Parameters passed include history of all exceptions thrown and how they were handled.

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
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 FINAL

For all $i < j \leq$ size:
values[i] $\preceq$ values[j]
**Lemma 1: ASSERT INNER, false => ASSERT INNER**

procedure \text{sort}(\text{values, size});
declare \text{values} real array[1000], \text{temp} real,
\text{i, j, size} integer;
\text{ASSERT INITIAL}
do for \text{i} = 1 to \text{size}-1
do for \text{j} = \text{i}+1 to \text{size}
if values[\text{j}] > values[\text{i}]
\text{temp} := values[\text{i}];
values[\text{i}] := values[\text{j}];
values[\text{j}] := \text{temp};
\text{ASSERT INNER}
end do;
\text{ASSERT OUTER}
end do;
\text{ASSERT FINAL}
end sort;

Symbolic Execution:

<table>
<thead>
<tr>
<th>Symb. Vals.</th>
<th>Path Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>values[\text{j}] = \text{IN} \text{1} 1</td>
<td>values[\text{i}] \leq \text{IN} \text{1} 2</td>
</tr>
</tbody>
</table>

**Symbolic Execution:**

- Symb. Vals: values[\text{j}] = \text{IN} 1
- Path Conditions: values[\text{i}] \leq \text{IN} 2
**Lemma 1: ASSERT INNER, false ⇒ 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**
value[i]' ≥ value[k]' for all i < k ≤ j

Symbolic Execution:
Symb. Vals. Path Conditions
values[i]' ≥ value[k]' for all i < k ≤ j

Thus:
values[i]' ≥ value[k]' (because IN₁ ≤ IN₂)

**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**
value[i]' = value[i] for all i < k ≤ j

Symbolic Execution:
Symb. Vals. Path Conditions
value[i]' = value[i] for all i < k ≤ j

Thus:
value[i]' = value[i] (because IN₁ ≤ IN₂)

Copyright L. Osterweil, all rights reserved
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[j];
  values[j] = values[i];
  values[i] = temp;

ASSERT INNER:
values[j] > values[k] for all i < k ≤ j

ASSERT OUTER:
values[j] > values[k] for all i < k ≤ j-1

end sort;
Lemma 2: \textbf{ASSERT INNER, true $\Rightarrow$ ASSERT INNER}

\begin{itemize}
  \item procedure \texttt{sort(values, size)};
  \item declare values \texttt{real array[1000]}, temp \texttt{real},
  \item \texttt{i}, \texttt{j}, size integer;
  \item \textbf{ASSERT INITIAL}:
  \item do for \texttt{i = 1 to size-1}
  \item do for \texttt{j = i+1 to size}
  \item \textbf{ASSERT INNER}:
  \item \texttt{values[i] $\geq$ values[j] for all $i < k \leq j$}
  \item Symbolic Execution:
  \item Symb. Vals. \texttt{Path Conditions}
  \item \texttt{values[i]} $\geq$ \texttt{values[j]}; \texttt{values[i]} $\geq$ \texttt{values[k]}; \texttt{values[i]} $\geq$ \texttt{values[l]}; \texttt{values[i]} $=$ \texttt{IN}; \texttt{values[j]} $=$ \texttt{IN}; \texttt{values[k]} $=$ \texttt{IN}; \texttt{values[l]} $=$ \texttt{IN};
  \item Thus:
  \item \texttt{value(i)} $>$ \texttt{value(j)}
  \item (because \texttt{IN} $>$ \texttt{IN})
  \item \textbf{ASSERT OUTER}:
  \item \textbf{ASSERT FINAL}:
\end{itemize}

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);
- assume \texttt{P}$\leq$\texttt{Q}, \texttt{Q}$<\texttt{E}$
Declare \texttt{P}, \texttt{Q}, \texttt{E}, \texttt{Y}, \texttt{A}, \texttt{B}, \texttt{D} real;
\texttt{A} $:= 0.0$; \texttt{B} $:= \texttt{Q} / 2.0$; \texttt{D} $:= \texttt{1.0}$; \texttt{Y} $:= 0.0$;
Do While (\texttt{D}$\geq$\texttt{E})
  If (\texttt{P} $-$ \texttt{A} $-$ \texttt{B} $\geq$ \texttt{0.0}) then (\texttt{Y} $:= \texttt{Y} + \texttt{D} / 2.0$; \texttt{A} $:= \texttt{A} + \texttt{B}$);
  \texttt{B} $:= \texttt{B} / 2.0$; \texttt{D} $:= \texttt{D} / 2.0$;
End do;
End Wensley;
What does Wensley’s algorithm do?

- Approximating \( P/Q (=Y) \) with error \( \leq E \)
- On the \( k \)th iteration of the loop
  
  \[
  A_k = a_1Q^{2^k-1} + a_2Q^{2^k-2} + \cdots + a_kQ^{2^k}
  \]
  
  \( a_i \in \{0,1\} \)
  
  \[
  B_k = Q^{2^k}
  \]
  
  \[
  D_k = 2^k
  \]
  
  \[
  Y_k = a_12^{2^k-1} + a_22^{2^k-2} + \cdots + a_k2^{2^k}
  \]
  
  \( a_i \in \{0,1\} \)

- \( P-A-B \geq 0 \) says when to add \( 2^{2^{k-1}} \) to \( Y_k \) ( \( a_{k+1} = 1 \))

- \( B_k = 2^{2^{k+1}} \cdot Q \)
**Assertions**

Initial:  \( A_s : \left(0 \leq P < Q \right) \land \left(0 < E\right) \)

Final:  \( A_f : \left(\left(\frac{P}{Q} - E\right) < Y \leq \frac{P}{Q}\right) \)

Intermediate:  \( A_i : \left(\left(\frac{A}{Q} \cdot Y \right) \cdot \left(\frac{B}{Q} \cdot \left(\frac{D}{2}\right)\right) \right) \land \left(2^k, k \text{ integer}, D = 2^k\right) \land \left(\left(\frac{P}{Q} - D\right) < Y \leq \frac{P}{Q}\right) \)

**Summary of Four Lemmas Needed**

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

**Lemmas called verification conditions**

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

\[ A_s : \left(0 \leq P < Q \right) \land \left(0 < E\right) \]

Input \( P, Q, E \)

\[ A \leftarrow 0.0 \]
\[ B \leftarrow Q/2 \]
\[ D \leftarrow 1.0 \]
\[ Y \leftarrow 0.0 \]

**Proof:**

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

**Lemma I: \( A_i \) to \( A_f \)**

\[ A_i : \left(\left(\frac{A}{Q} \cdot Y \right) \cdot \left(\frac{B}{Q} \cdot \left(\frac{D}{2}\right)\right) \right) \land \left(2^k, k \text{ integer}, D = 2^k\right) \land \left(\left(\frac{P}{Q} - D\right) < Y \leq \frac{P}{Q}\right) \]

Input \( P, Q, E \)

\[ A \leftarrow 0.0 \]
\[ B \leftarrow Q/2 \]
\[ D \leftarrow 1.0 \]
\[ Y \leftarrow 0.0 \]

\[ A_i \]
**Lemma II: A_i, false branch, A_i**

A_i:
\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \text{ for some integer } k \]
\[ P/Q - D < Y \leq P/Q \]

\[ D \geq E \text{ [constraint]} \]
\[ P - A - B < 0 \text{ [constraint]} \]
\[ B = B/2 \]
\[ D = D/2 \]

\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \]
\[ P/Q - D < Y \leq P/Q \]

**Proof of Lemma II**

(Code execution shows)

A_i = A, B_i = B/2; D_i = Y = Y_i

Proof:
1) \[ A_i = A = Q \cdot Y = Q \cdot Y_i \]
2) \[ B_i = B/2 = (Q \cdot D/2)/2 = Q \cdot D/2 \]
3) \[ D_i = D/2 = 2/2 = 2^{k-i} \]
4) a) \[ P - A_i - B_i < 0 \text{ (constraint)} \]
\[ P/Q - D_i < Y \leq P/Q \]
\[ B = B/2 \]
\[ D = D/2 \]

**Lemma III: A; True branch; A_i**

A_i = Q \cdot Y
B_i = Q \cdot D/2
D_i = 2^k \text{ for some integer } k
P/Q - D < Y \leq P/Q

**Proof of Lemma III**

From symbolic execution we know:
\[ A_i = A + B; \quad B_i = B/2; \quad D_i = D/2; \quad Y_i = Y + D/2 \]

We also know:
\[ P - A_i - B_i < 0 \quad P/Q - D_i < Y \]

Proof:
1) \[ A_i = A + B = Q \cdot Y + Q \cdot (D/2) = Q \cdot (Y + D/2) = Q \cdot Y_i \]
2) \[ B_i = B/2 = Q \cdot D/2 = Q \cdot D_i/2 \]
3) \[ D_i = D/2 = 2^{k-i} = 2^{k-1} \]
4) a) \[ P - A_i - B_i < 0 \quad P/Q - D_i < Y \]
\[ P/Q - D_i < Y \leq P/Q \]
\[ B = B/2 \]
\[ D = D/2 \]

**Lemma IV A_i, A_f**

A_i, false, A_f

[Diagram showing the relation between A_i and A_f]
**Lemma IV**

\[ A_i: (A=Q^i Y) \land (B=Q^{i+1}(D/2)) \land (P>Q^i D) \land (P/Q-D)<Y \leq P/Q \]

Proof:

\[ \text{Given } ((P/Q)-D)<Y \leq (P/Q) \text{ and } (D < E) \Rightarrow ((P/Q)-E)<((P/Q)-D)<Y \leq (P/Q) \]

**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

**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**

- 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

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? .....)

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
  - 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?