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....
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)

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

Example program

```plaintext
procedure Contrived
begin
X, Y, Z: integer;
read X, Y;
if X - Y < 0 then
endif;
else
Z := X+Y;
if X ≥ 3 then
endif;
 write Z;
end Contrived;
end Contrived;
```

Note: "X_i" means the ith value taken as input to the program
procedure Contrived is
  X, Y, Z : integer;
  begin
    if X ≥ 3 then
      Z : = 0;
    else
      if Y > 0 then
        Y := Y + 5;
      end if;
      if X - Y < 0 then
        Z := X + Y;
      else
        if X = 3 then
          Y := Y + 5;
        else
          write Z;
          end if;
        end if;
    else
      if Y > 0 then
        write Y;
      end if;
    end if;
    read X, Y;
    Z := 0;
  end Contrived;

Results (feasible path)

Results (infeasible path)

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...

\[ P = 1, 2, 3, 5, 6, 7, 9 \]
\[ D[P] = \{ (ln_1, ln_3) \mid ln_2 \leq 0 \land (ln_1, ln_2) \geq 5 \} \]
\[ C[P] = PV.Y = ln_2 + 5 \]
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 (e.g., 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
  - Checking relations between code and design
- Comparison: Runtime evaluation of assertions
  - Facilities for programming reactions to violations
  - Also useful as a debugging aid

Assertion-Based Dynamic Testing

- Specification of Intended Behavior
- Functional Behavior Assertions
- Intermediate Execution Results
- Reports on Internal Failures
- Comparison of Behavior to Intent
**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.
  - 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.

---

**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 Cₙ**

**Theorem:**

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

then $|E_n| = n \cdot \frac{(n-1)}{2}$

---

**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**

- 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)
- So, \( |E_{n+1}| = \frac{n(n-1)}{2} + n \)
  \(-= \frac{n(n+1)}{2} = \frac{n(n+1)}{2} \)
  \(-= \frac{(n+1)(n+1)}{2} \)
  \(-= \frac{(n+1)n}{2} \)

**Floyd’s Method of inductive assertions**

- 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 assertion free code
  - and end with an assertion
- eg. (\( A_0 \) =) \( A_0, C_1, A_1, C_2, A_2, \ldots A_{n-1}, C_{n-1}, A_n, C_n, A_{n+1}(=A_f) \)
- 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**

![Diagram showing straight-line code, initial assertion, intermediate assertions, and final assertion.]

**Must be sure:**

assuming \( A_0 \), then executing Code \( C_n \),

necessarily \( \Rightarrow \) \( A_{i+1} \)

by forward substitution

\[ A_0 \quad C_i \quad A_{i+1} \]

**Why does this work?**

suppose \( P = \) arbitrary path through the program can denote it by

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

where

- \( A_0 \) - 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

\[ A_0, \ldots, A_{i+1} \]
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

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;

What does Wensley's algorithm do?

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

- Since $0 \leq P/Q < 1$, then $P/Q$ can be estimated as a sum of the series
  \[ a_2 2^{-1} + a_3 2^{-2} + \ldots + a_k 2^{-k} \]
  \[ a_i \in \{0,1\} \]

- 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 error

- $P(A_k + B_k)$ says when to add $2^{(k+1)}$ to $Y_k$, ($a_{k+1} = 1$)

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

### Assertions

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

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

**Intermediate:**

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

**Input:** $P$, $Q$, $E$

**Output:** $A$, $B$, $D$, $Y$

**Summary of Four Lemmas Needed**

- **I.** Initial assertion, $A_0$, to $A_1$
- **II.** $A_i$, false branch, to $A_i$
- **III.** $A_i$, true branch, to $A_i$
- **IV.** $A_i$, to $A_F$, final assertion

### Lemma I: $A_0$ to $A_1$

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

**Proof:**

1. $A = 0; Q \cdot Y = Q \cdot 0 = 0$
2. $A = Q \cdot Y$
3. $B = Q/2$; $D = 1$; $Y = 0; 0$

**Input:** $P$, $Q$, $E$

**Output:** $A$, $B$, $D$, $Y$

### Lemma II: $A_i$, false branch, $A_i$

**Lemma II:** \[ A_i: \{(P/Q - D) < Y \leq (P/Q)\} \]

**Proof:**

1. $D = 2^k$ for some integer $k$
2. $P - A \cdot B < 0$ (constraint)
3. $B \leftarrow B/2$
4. $D \leftarrow D/2$

**Input:** $P$, $Q$, $E$

**Output:** $A$, $B$, $D$, $E$
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:
  - \( A' = A, B' = B/2; D' = D/2; Y' = Y \)

From symbolic execution we know:

- \( A'' = A + B; B'' = B/2; Y'' = Y + D/2 \)
- \( D'' = 2^k \) for some integer \( k \)
- \( P/Q - D < Y \)
- \( P - A - B \geq 0 \) and \( D \geq E \)

Proof of Lemma II

(Symbolic execution shows)

\[ A = Q \cdot Y \]
\[ B = Q \cdot D/2 \]
\[ D = 2^k \] for some integer \( k \)
\[ P/Q - D < Y \]
\[ D \geq E \] [constraint]
\[ P - A - B \geq 0 \] [constraint]
\[ B = B/2 \]
\[ D = D/2 \]

Proof:

1) \( A' = A = Q \cdot Y \)

2) \( B' = B/2 \) (uses \( Q > 0 \))

3) \( D' = D/2 < Y \)

4) \( A' = Q \cdot Y' \)

5) \( B' = Q \cdot D'/2 \)

6) \( D' = 2^k \) for some integer \( k \)

\[ P/Q - D' < Y \]

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)

- \( P/Q - D < Y \)
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

- Unsuccessful proof attempt $\rightarrow$ ???
  - 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→ 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 (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

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 ...

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

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?