More Verification

Reading assignment

- L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," **ACM Computing Surveys**, 8 (3), September 1976, pp. 306-330. (not required)
- K. M. Olender and L. J. Osterweil, "Interprocedural Static Analysis of Sequencing Constraints," **ACM Transactions on Software Engineering and Methodology**, 1 (1), January 1992, pp. 21-52.

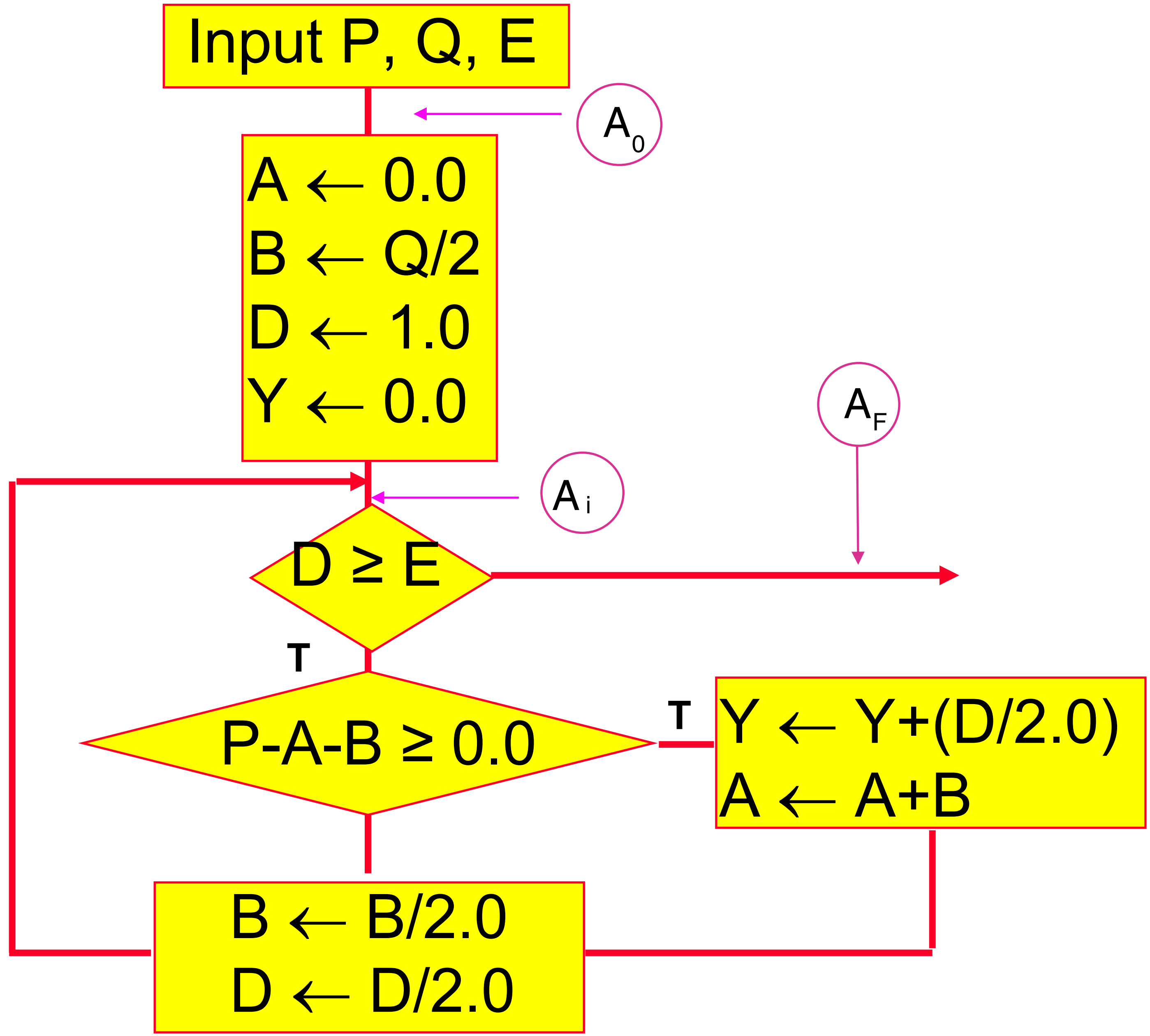
Floyd's Inductive Verification Method

- Specify initial and final assertions to capture intent
- Place intermediate assertions so as to "cut" every program loop
- For each pair of assertions where there is at least one executable (assertion-free) path from the first to the second,
  - assume that the first assertion is true
  - show that for all (assertion-free, executable) paths from the first assertion to the second, that the second assertion is true
- This above establishes "partial correctness"
- Show that the program terminates
  - This establishes "total correctness"

Wensley's Algorithm

```
Procedure Wensley (P: input, Q: input, E: input, Y: output)
--assume 0≤ P<Q, 0< E
-- approximating P/Q (=Y) with error ≤ 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;
```

Flow Graph



Input P, Q, E
A0
A ← 0.0
B ← Q/2
D ← 1.0
Y ← 0.0
AF
Ai
D ≥ E
T
P-A-B ≥ 0.0
T
Y ← Y+(D/2.0)
A ← A+B
B ← B/2.0
D ← D/2.0

What does Wensley's algorithm do?

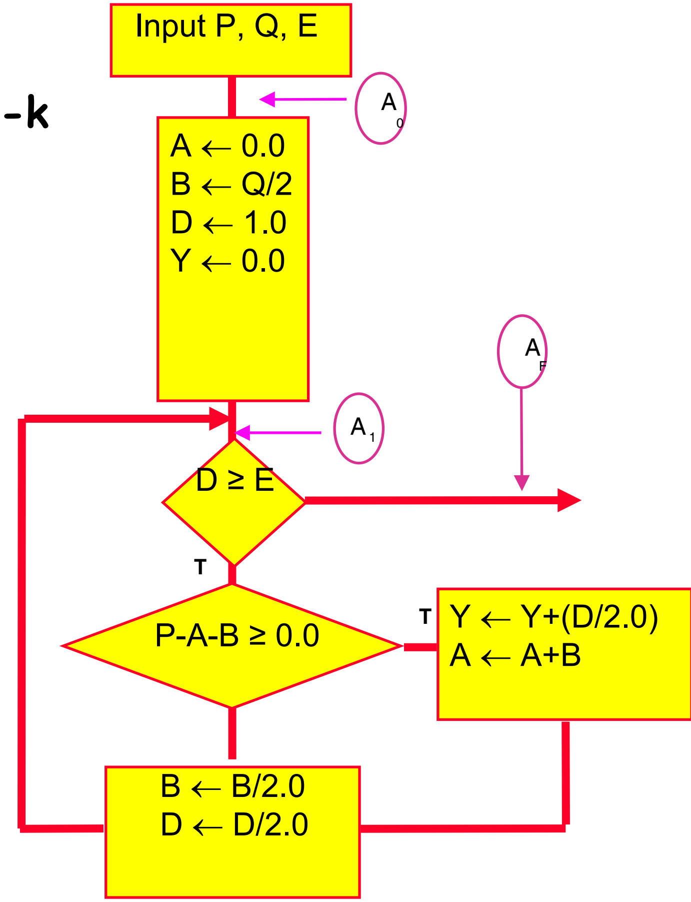
- approximating P/Q (=Y) with error ≤ E
- on the kth iteration of the loop

$$Y_k = c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + \ \dots \ + c_k \cdot 2^{-k}$$
$$c_i \in \{0,1\}$$

$$A_k = c_1 Q \cdot 2^{-1} + c_2 Q \cdot 2^{-2} + \dots + c_k \cdot Q 2^{-k}$$
$$c_i \in \{0,1\}$$
$$= Q \cdot Y_k \approx P$$

$$B_k = Q \cdot 2^{-k} \text{ next term}$$

$$D_k = 2^{-k}$$



What does Wensley's algorithm do?

- since $0 \leq P/Q < 1$, then P/Q can be estimated as a sum of the series

$$c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + \ldots + c_k \cdot 2^{-k}$$
$$c_i \in \{0,1\}$$

  - Y_k is the computed value of the quotient
  - given $Y_k Q = A_k$ shows how close the computed quotient is to the real quotient
  - D_k is the computed error
  - $P-(A_k + B_k)$ then add $2^{-(k+1)}$ to Y_{k+1} (by setting $c_{k+1} = 1$)

Assertions

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

computed quotient

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

computed error

Intermediate:

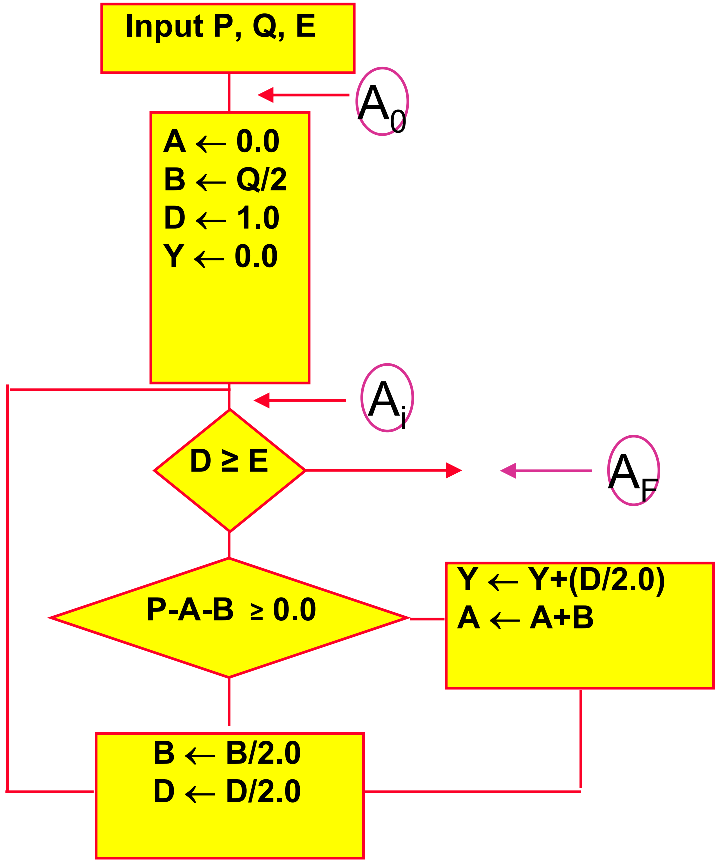
A_i: $\{(A = Q*Y) \wedge (B = Q*(D/2))$

$\wedge\ (k \geq 0,\ k \text{ integer} \wedge D = 2^{-k})$

$\wedge\ ((P/Q) - D) < Y \leq (P/Q)\}$

Y is within the computed error D of P/Q

Summary of Four Lemmas Needed



I. Initial assertion to A_i
II. A_i, false branch, A_i
III. A_i, true branch, A_i
IV. A_i, final assertion

Lemmas called verification conditions
$0 \leq k$ is the number of completed iterations

Lemma I: A_0 to A_i

A_0: Initial Assertion

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

code {
Input P, Q, E
A←0;
B←Q/2;
D←1;
Y←0;
}

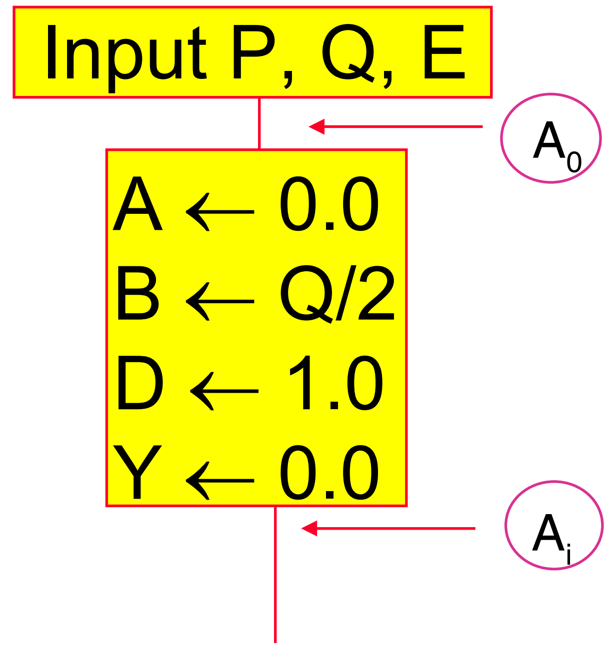
$\Rightarrow A_i$:

$A = Q * Y$

$B = Q * D/2$

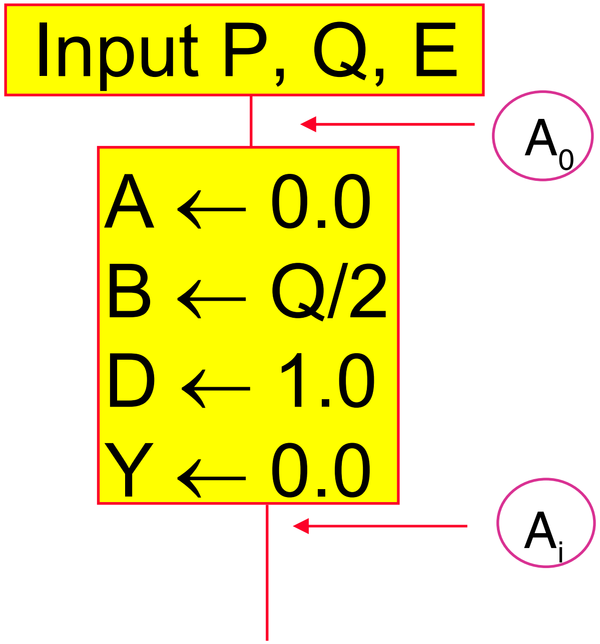
$D = 2^{-k}, \quad k = 0$

$P/Q - D < Y \le P/Q$



Note, $k = 0$

By symbolic execution (by s. e.)



A = 0;
B = Q/2;
D = 1;
Y = 0;

Lemma I: A_0 to A_i

A_0: Initial Assertion

$(0 \le P < Q) \wedge (0 < E)$

code
```
A←0;
B←Q/2;
D←1;
Y←0;
```

$\Rightarrow A_i$:

$A = Q * Y$

$B = Q * D/2$

$D = 2^{-k}$, $k=0$

$P/Q - D < Y \le P/Q$

Proof: Given Input P, Q, E

1) $A = 0$ (by s. e.)
$= Q * 0$ (rewrite)
$= Q * Y$ (by s. e.)

2) $B = Q/2$ (by s. e.)
$= Q * 1/2$ (rewrite)
$= Q * D/2$ (by s. e.)

3) $D = 1$ (by s. e.)
$= 2^{-0}$

4) $0 \le P < Q$ (given)
$\Rightarrow 0 \le P/Q < 1$ (divide by Q, Q>0)
$\Rightarrow P/Q - 1 < 0 \le P/Q$ (rewrite)
$\Rightarrow P/Q - D < Y \le P/Q$ (by s. e.)

Lemma II: A_i, false branch, A_i

A_i:

$A = Q * Y$

$B = Q * D/2$

$D = 2^{-k}$ for some integer k

$P/Q - D < Y \leq P/Q$

$D \geq E$

$P - A - B < 0$

$B \leftarrow B/2$

$D \leftarrow D/2$

} code

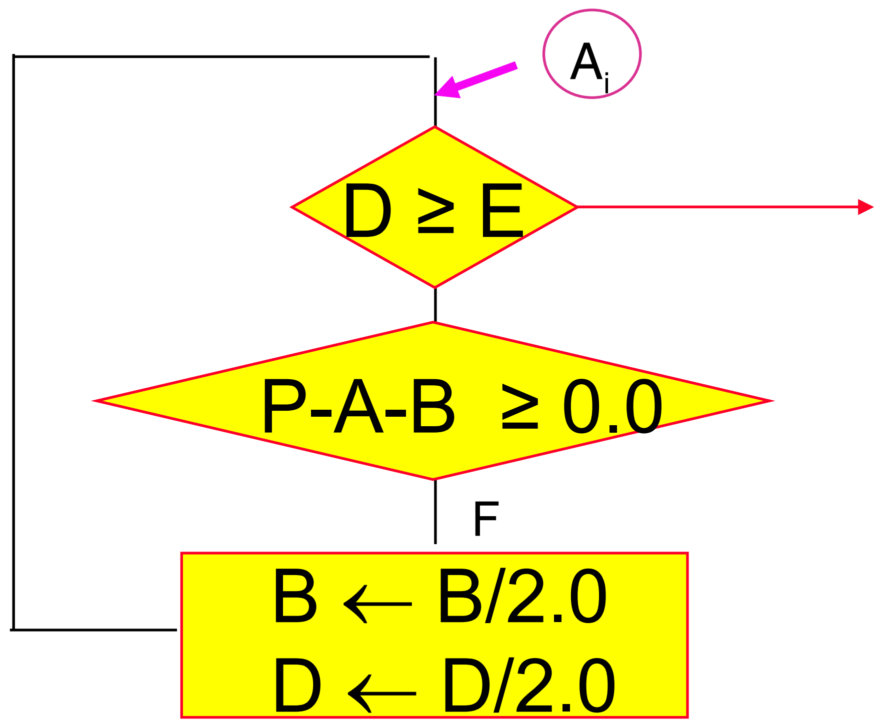
$\Rightarrow A_i$:

$A = Q * Y$

$B = Q * D/2$

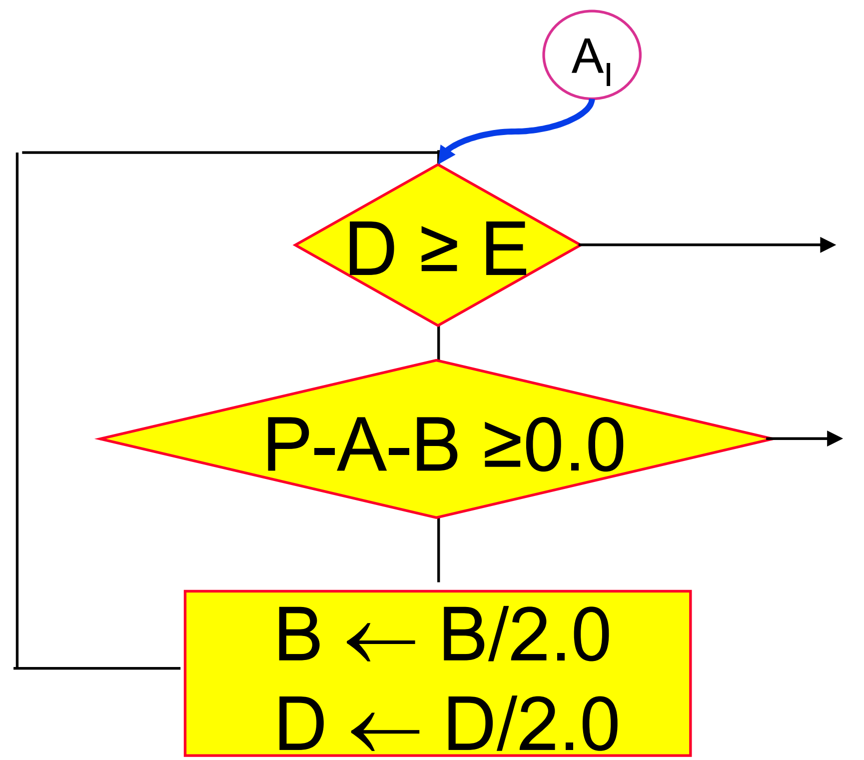
$D = 2^{-(k+1)}$

$P/Q - D < Y \leq P/Q$



Proof of lemma II

- **Need to establish that A_i is a correct after loop execution, based on assumption that A_i was correct before loop execution**
- **Notation:**
  - A, B, D, Y are original values of variables
  - B', D' are values after loop execution
- **Symbolic execution gives:**
  - **D ≥ E**
  - **P - A - B < 0**
  - **B'= B/2**
  - **D' = D/2**



Proof of Lemma II

$A = Q * Y$
$B = Q * D/2$
$D = 2^{-k}$ for some integer k
$P/Q - D < Y \le P/Q$

$D \ge E$
$P - A - B < 0$
$B \leftarrow B/2$
$D \leftarrow D/2$

} code

$A = Q * Y$
$B' = Q * D'/2$
$D' = 2^{-(k+1)}$ for some integer k
$P/Q - D' < Y \le P/Q$

(Symbolic execution shows
$D \ge E$; $P-A-B<0$; $B' = B/2$; $D' = D/2$)

Proof:

1) $A = Q * Y$ (given)

2) $B' = B/2$ (by s. e.)
$= (Q * D/2)/2$ (by given)
$= (Q * D'/2)$ (by s. e.)

3) $D' = D/2$ (by s. e.)
$= 2^{-k}/2$ (by given)
$= 2^{-k-1}$ (rewrite)
$= 2^{-(k+1)}$ (rewrite)

Proof of Lemma II

$A = Q * Y$
$B = Q * D/2$
$D = 2^{-k}$ for some integer k
$P/Q - D < Y \leq P/Q$

$D \geq E$
$P - A - B < 0$
$B \leftarrow B/2$
$D \leftarrow D/2$

$A = Q * Y$
$B' = Q * D'/2$
$D' = 2^{-(k+1)}$ for some integer k
$P/Q - D' < Y \leq P/Q$

(Symbolic execution shows
$B' = B/2$; $D' = D/2$; $D \geq E$; $P - A - B < 0$)

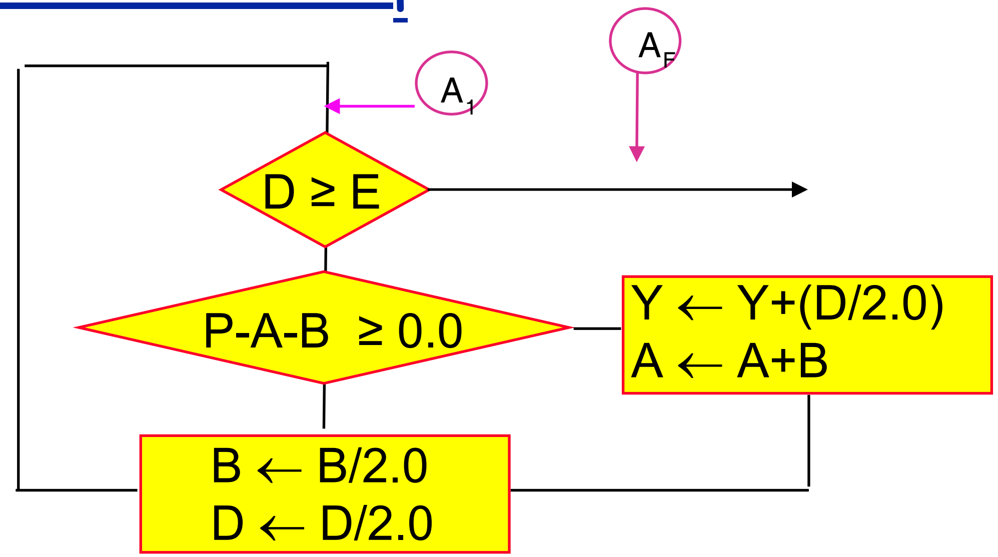
Proof (continued):

4)
a) $P-A-B < 0$ (by s. e.)
=> $P - Q * Y - Q * D/2 < 0$ (by given)
=> $P/Q - Y - D/2 < 0$ (divide by $Q > 0$)
=> $P/Q - D/2 < Y$ (rewrite)
=> $P/Q - D' < Y$ (by s. e.)

b) $Y \leq P/Q$ (given)
=> $Y' \leq P/Q$ (by s. e.)

Lemma III: A_i; True branch; A_i

Ai: $A = Q * Y$
$B = Q * D/2$
$D = 2^{-k}$ for some integer k
$P/Q - D < Y \le P/Q$



$D \ge E$
$P - A - B \ge 0$
$Y \leftarrow Y+(D/2.0)$
$A \leftarrow A+B$
$B \leftarrow B/2$
$D \leftarrow D/2$

$\Rightarrow A_i$:
$A' = Q * Y'$
$B' = Q * D'/2$
$D' = 2^{-(k+1)}$ for some integer k
$P/Q - D' < Y' \le P/Q$

From symbolic execution we know:

$A' = A + B$; $B' = B / 2$;
$D' = D / 2$; $Y' = Y + D / 2$;
$P - A - B \ge 0$; $D \ge E$

Proof Lemma III

Ai: $A = Q * Y$
$B = Q * D/2$
$D = 2^{-k}$ for some integer k
$P/Q - D < Y \leq P/Q$

$D \geq E$
$P - A - B \geq 0$
$Y \leftarrow Y+(D/2.0)$
$A \leftarrow A+B$
$B \leftarrow B/2$
$D \leftarrow D/2$

$\Rightarrow A_i$:
$A' = Q * Y'$
$B' = Q * D''/2$
$D' = 2^{-(k+1)}$ for some integer k
$P/Q - D'' < Y' \leq P/Q$

From symbolic execution we know:
$P - A - B \geq 0$; $D \geq E$; $A' = A + B$;
$B' = B / 2$; $D' = D / 2$; $Y' = Y + D / 2$

Proof:

1) $A' = A + B$ (by s.e.)
$= Q*Y + Q*(D/2)$ (by given)
$= Q(Y + D/2)$ (rewrite)
$= Q * Y'$ (by s.e.)

2) $B' = B/2$ (by s.e.)
$= Q * D/2/2$ (by given)
$= Q * D'/2$ (by s.e.)

3) $D' = D/2$ (by s.e.)
$= 2^{(-k-1)}$ (by given)
$= 2^{-(k+1)}$ for some K

Proof Lemma III

Ai: $A = Q * Y$
$B = Q * D/2$
$D = 2^{-k}$ for some integer k
$P/Q - D < Y \leq P/Q$

$D \geq E$
$P - A - B \geq 0$
$Y \leftarrow Y+(D/2.0)$
$A \leftarrow A+B$
$B \leftarrow B/2$
$D \leftarrow D/2$

$\Rightarrow A_i$:
$A' = Q * Y'$
$B' = Q * D'/2$
$D' = 2^{-(k+1)}$ for some integer k
$P/Q - D' < Y' \leq P/Q$

From symbolic execution we know:
$A' = A + B$; $B' = B / 2$;
$D' = D / 2$; $Y' = Y + D / 2$;
$P - A - B \geq 0$; $D \geq E$

Proof (continued):

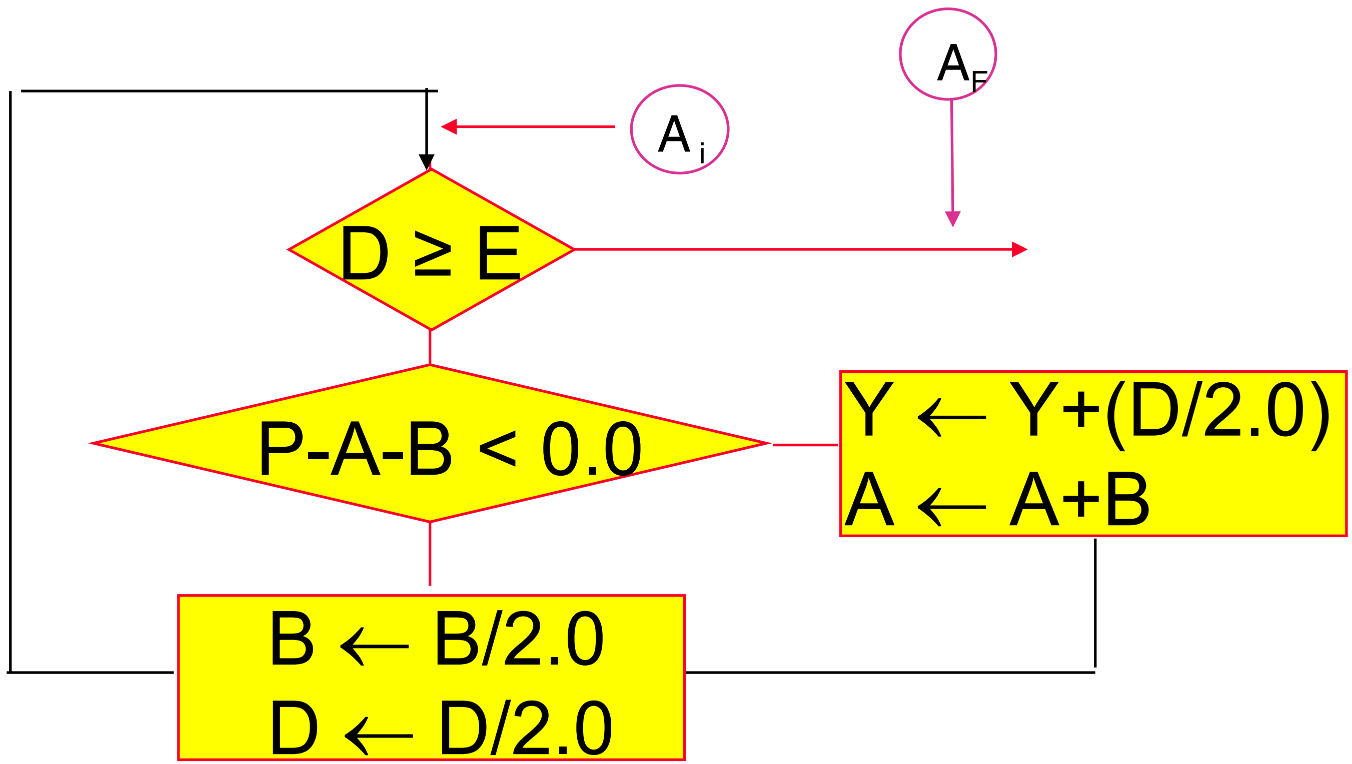
4)

a) $P - A - B \geq 0$ (by s.e.)
$\Rightarrow P - Q * Y - Q * (D/2) \geq 0$ (given)
$\Rightarrow P/Q - D/2 \geq Y$ (rewrite)
$\Rightarrow P/Q - D/2 \geq Y' - D/2$ (by s.e.)
$\Rightarrow P/Q \geq Y'$ (rewrite)

b) $P/Q - D < Y$ (given)
$\Rightarrow P/Q - D < Y' - D/2$ (by s.e.)
$\Rightarrow P/Q - D/2 < Y'$ (rewrite)
$\Rightarrow P/Q - D' < Y'$ (by s.e.)

Lemma IV A_i, A_F

- A_i, false, A_F



Lemma IV

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

$D < E$] code

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

Proof:

$((P/Q)-D)<Y\leq(P/Q)$ and $(D < E)$ (given and s.e.)

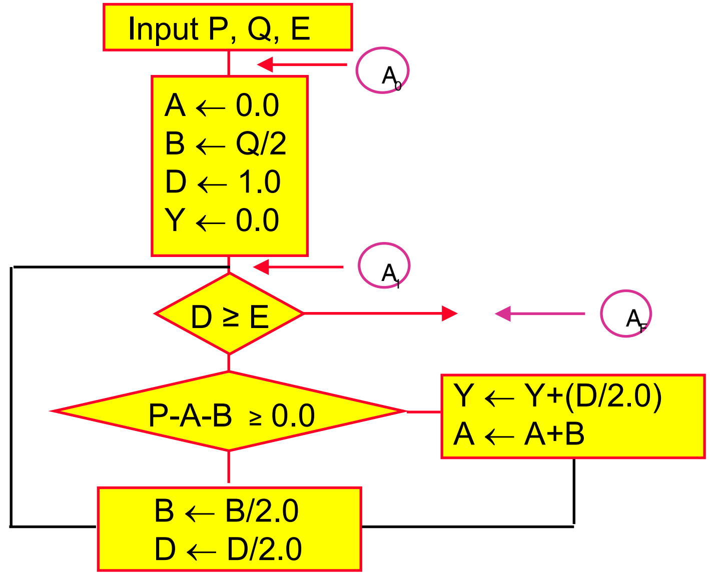
$\Rightarrow$ $((P/Q)-E)<((P/Q)-D)<Y\leq(P/Q)$ (rewrite)

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

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 decreases
        on each iteration and
        E does not change
    Thus, eventually D<E
        and the loop terminates



Social Processes and Proofs of Theorems and Programs

- by Richard DeMillo, Richard Lipton, and Alan Perlis -CACM May 1979
- controversial paper
  - changed funding program in U.S
  - almost halted verification research
- verification community was guilty of overselling their product
- some say that the paper went overboard in refuting the claims of the verification community

What was the motivation?

- verification community hyperbole was negatively affecting other research
  - language design
    - e.g. Euclid did not include exception handling because there would not be any run time errors
  - Testing & Analysis
    - any method that provides partial information was rejected as unnecessary
      - symbolic execution
      - testing

On the other hand

- Verification had a very positive impact on software engineering
  - major argument for structured programming
    - Djkstra'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 data structures
    - input/output assertions for all operations

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

Formal verification process

- Proofs of programs are not interesting and therefore will not go thru this social process
- Automatic verification is not feasible for most programs
  - Search space is too large
  - Need additional axioms, which will not be "socially" accepted

Specification Problem

- real programs are not captured by simple mathematical algorithms
  - 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

Verification Successes

- Model Checking
  - IEEE future bus
  - ISDN User Part Protocol
  - HDLC (data link controller)
- Theorem Proving
  - SRT division algorithm
  - Motorola 68020 (compiler code generation)
  - AMD5K86 (floating point division)

Is Proof More Cost-Effective than Testing?

- TSE, August 2000
- King, Hammond, Chapman, and Pryor
- Praxis Critical Systems
- Case Study
  - Ship Helocopter Operating Limits Information Systems (SHOLIS)
  - Safety critical system
    - Must conform to UK DoD safety critical standards

Software System

- Written in a subset of Ada (called SPARK)
- Annotations for describing pre, post, assert, and return assertions
- Restrictive programming style
  - No user-defined exceptions, aliasing (?), go to's,functions with side-effects, recursion, generics, tasks

Development process

- Requirements written in English, not s/w related
- Software Requirements Spec (SRS) written in Z and English
  - 300 pages
- Software Design Spec (SDS) written in Z, English and some SPARK
  - Added implementation details
  - 200 pages ?
- Code written in SPARK

Code annotations

- Assertions
  - Pre, post, assert, return
- Additional info
  - Global, derives, own, and inherit
  - Extends Ada typing
  - Checked by Spark tools
- Z used to define annotations

Code

- 133 KLOCS
  - 13K declarations
  - 14K executable stmts
  - 54K annotations
  - 20K SPARK proof annotations
  - 32K blank or comments lines

Z proofs at the SRS and SDS level

- Proof by rigorous arguments with some automated assistance
- 150 Proofs, about 500 pages
- Add and proved safety properties

Code proofs

- Automated
- Examiner--creates the verification conditions
  - 9000 verification conditions
- Simplifier
  - discharged 76% of the verification conditions
- Proof Checker
  - discharged most of the remaining 24%
  - Some discharged by informal justification
- Proved all loops terminiated

Fault detection

Validation phase	% faults found	% Effort
Specs	3.25	5
Z proof	16	2.5
HL design	1.5	2
LL design	26.25	17
Unit test	15.75	25
Integration	1.25	1
Code proof	5.25	4.5
System test	21.5	9.5
Acceptance	1.25	1.5
Other	8	32

Overall

- 19 person year effort
- Lessons learned
  - Limits to formality
  - Top level proofs too large
    - Only did important safety properties
  - Low level proofs interact with outside devices
  - Target compiler ran under different assumptions than the SPARK compiler
    - E.g., memory management and floating pt.
  - Claim: Using proofs led to a simpler system design

Observations about Formal Verification

- Most proofs are simple but some proofs are long, tedious & hard
- assertions are hard to get right
- invariants are difficult to get right
  - need to support overall proof strategy
- proofs themselves often require deep program insight
  - Often require axioms about the domain

Deeper Issues

- unsuccessful proof attempt ⇒ ???
  - incorrect software
  - incorrect assertions
  - incorrect placement of assertions
  - inept prover
  - any combination (or all) of the above

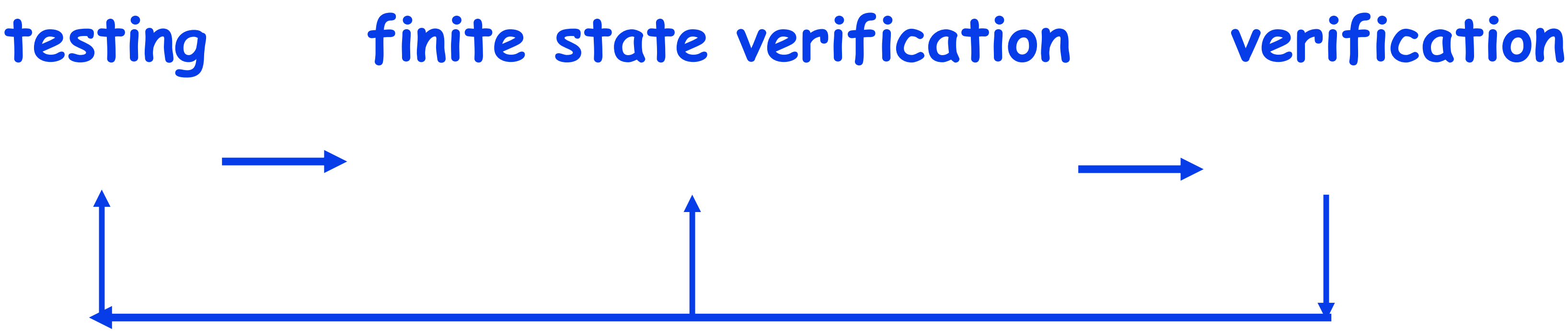
although failed proofs often indicate which of the above is likely to be the problem (especially to an astute prover)

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

Current Status:

- have verified some non-trivial programs or important parts of programs
  - e.g., protocol verification, SHOLIS
- improved theorem provers
- improved specification languages
- verification and testing/analysis research now viewed more as a continuum



Current Status

- Software systems are becoming
  - More complex
  - Distributed
- need:
  - Good people that are well-trained
  - Techniques that good people can use
- Research trends
  - Finite state verification for well-trained practitioners
  - Finite state verification combined with theorem proving based verification