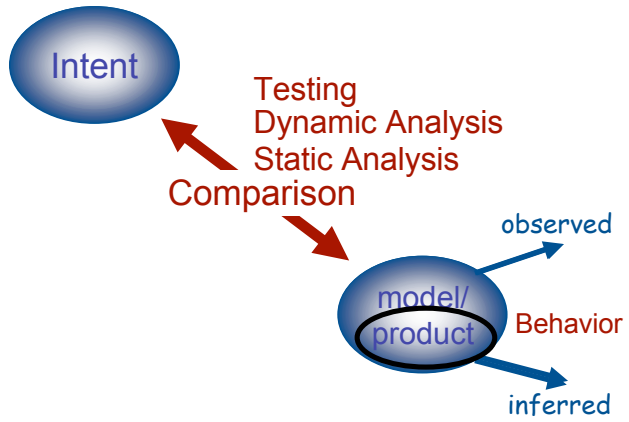
Announcements

- Revised Project 2 (minor) and new Project 3 posted
  - Note: each group (on-campus) needs to arrange a design review during the period 11/29-12/7
  - **No class on 12/6** (can be used for design review)
  - Due Date for Project 2 extended to 12/8
- As alternatives to Rational Rose, I have obtained licenses for Eclipse & Visual Paradigm; licenses are (will be available) on the website.
- I will have comments/grades on Project 1 on 11/29

22-Analysis Overview

- Readings:
  - GJM03 Chapter 6
  - Ost76 Osterweil, Leon J. and L.D. Fosdick, " DAVE--A Validation Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, September 1976, Vol. 6., pp. 473-486
  - Ole92 Olender, K. M. and L. J. Osterweil, "Interprocedural Static Analysis of Sequencing Constraints," ACM Transactions on Software Engineering and Methodology, January 1992, 1(1), pp. 21-52.
  - Dwy95 Dwyer , M. B. and Clarke, L. A. ÒA Flexible Architecture for Building Data Flow Analyzers," in CMPSCI Technical Report, August 17, 1995
  - Adr82 Adrion, W.R,. M.A. Branstad, and J.C. Cherniavsky, "Validation, Verification and Testing of Computer Software," ACM Computing Surveys, June 1982, pp.159--192
  - Hoa69 Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, October 1969.
  - Flo67 Floyd, R.W. "Assigning Meaning to Programs", in the Proceedings of Symposium on Applied Mathematics, 1967, pp. 19-32, (Appeared as volume 19 of Mathematical Aspects of Computer Science).
  - Han76 Hantler, S.L. and J.C King,. "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, September 1976, pp. 278-300.
  - Cla85 Clarke L. A. and D. J. Richardson, "Applications of Symbolic Evaluation," Journal of Systems and Software, January 1985, 5 (1), pp.15-35.
  - Zhu97 Zhu, Hong, Patrick A. V. Hall, and John H. R. May, "Software Unit Test Coverage and Adequacy," ACM Computing Surveys, vol. 29, no.4, pp. 366-427, December, 1997.
  - Wey80 Weyuker, Elaine J. and Thomas Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, May 1980, SE-6(5), pp. 236-246
  - DeM79 DeMillo R.A. and R.J. Lipton and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs, Communications of the ACM, May 1979, 22(5), pp. 271-280

Analysis



Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

Basic Verification Strategy

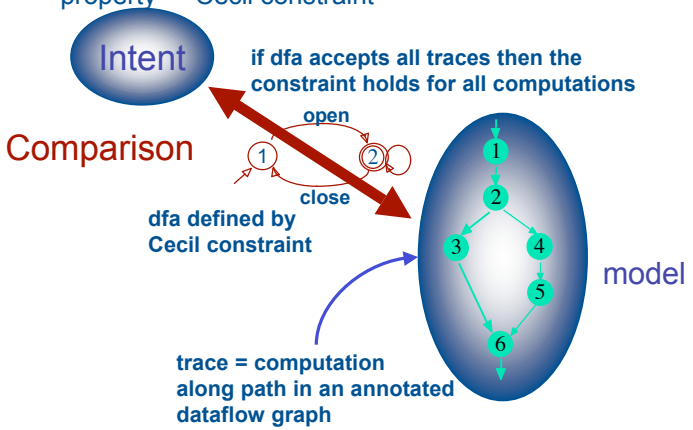
- analyze a system for desired properties, i.e., compare behavior to intent
  - intent
    - can be expressed as properties of a model
    - can be expressed as formulas in mathematical logic
  - behavior
    - can be observed as software executes
    - can be inferred from a model
    - can be expressed as formulas in mathematical logic
  - different representations support different sorts of inferences
    - comparison can be informal
      - done by human eye, e.g., inspection
    - can be done by computers
      - comparing text strings
    - can be done by model-checkers
      - such as formal machines (e.g., fsa's)
    - can be done by rigorous mathematical reasoning

Example: Dataflow Analysis

- intent:
  - stated as a property
  - captured as an event sequence
- behavior:
  - model represents some execution characteristics
  - inferred from a model: (e.g., annotated flow graph)
  - inferences based upon:
    - semantics of flow graph
    - semantics captured by annotations
- comparison:
  - done by a fsa (e.g., a property automaton)

Data FlowAnalysis

"property" = Cecil constraint



Data Flow Analysis (DFA)

- Uses an annotated control flow graph model of the program
  - Compute facts for each node
  - Use the flow in the graph to compute facts about the whole program
- DFA used extensively in program optimization, e.g.,
  - determine if a definition is dead (and can be removed)
  - determine if a variable always has a constant value
  - determine if an assertion is always true and can be removed
- Some Dataflow systems
  - DAVE system demonstrated the ability to find def/ref anomalies
  - Cecil/Cesar system demonstrated the ability to prove general user-specified properties
  - FLAVERS demonstrated applicability to concurrent system

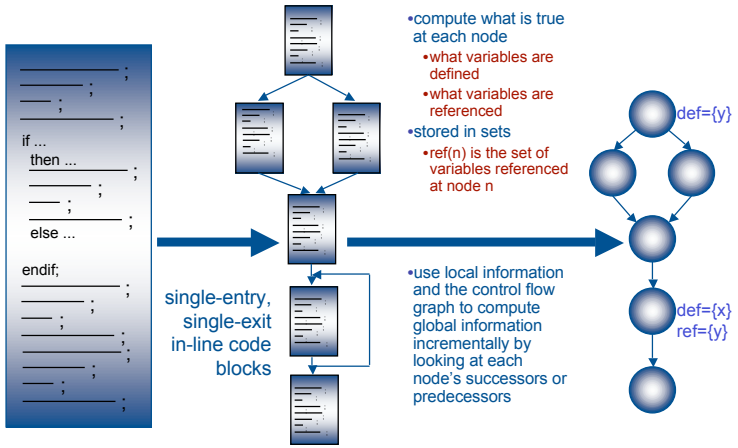
COMPUTER SCIENCE

Data flow analysis

- computes information that is true at each node in the CFG, e.g.,
  - what variables are defined
  - what variables are referenced
- usually stored in sets
  - ref(n) is the set of variables referenced at node n
- uses this local information and the control flow graph to compute global information about the whole program
  - done incrementally by looking at each node's successors or predecessors

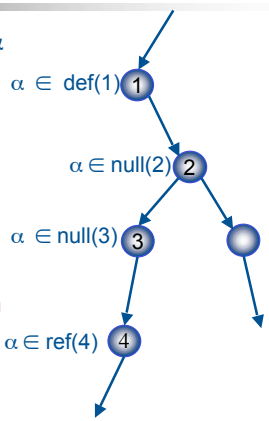
COMPUTER SCIENCE

DFA



Def-ref path expressions

- for a path P and a variable α can write a path expression describing the sequence of set memberships encountered for a, where
  - α ∈ def (n) or
  - α ∈ ref(n) or
  - α ∈ null (n)
  - for each node n on the path
- write (and simplify) a path expression
  - $P(n_1, n_1, \ldots, n_1; \alpha)$



P(1,2,3,4; α) = d11r = dr

Anomalous pairs of ref/defs

d - defined, r - referenced, u - undefined

dd	bug?	du	bug?
dr	normal	ud	normal
uu	harmless?	rr	normal
rd	normal	ru	normal
ur	bug		

unreferenced definition

undefined reference

Consider unreferenced definition

- Want to know if a def is not going to be referenced
  - dd or du
- At the point of a definition of a, want to know if there is some path where a is defined or undefined before being used
  - May be indicative of a problem if the path is executable
  - Usually just a programming convenience and not a problem
- At the point of a definition of a, want to know if on all paths a is defined or undefined before being used
  - May be indicative of a problem
  - Or could just be wasteful

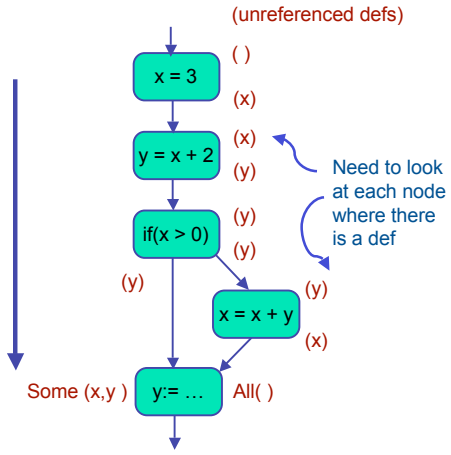
global dataflow analysis

- classes
  - forward flow problems (e.g., available expressions)
    - what definitions can affect computations at a given point in a program
  - backward flow problems (e.g., live variables)
    - what uses that follow a given point in the program can be affected by computations up to that point
- paths
  - any path
  - all path

Unreferenced definitions

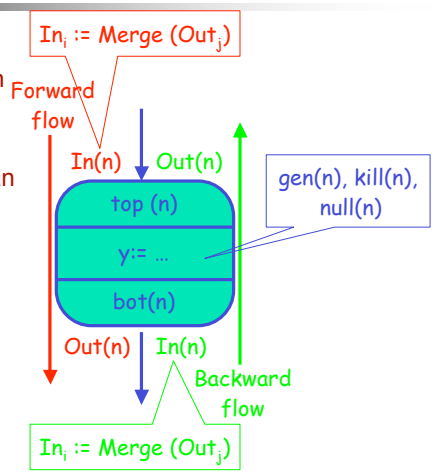
```
int x,y;
...
x := 3;
y := x + 2;
if x > 0 then
  x := x + y;
end if;
y:= …
```

Forward flow,
all paths problem



General Approach

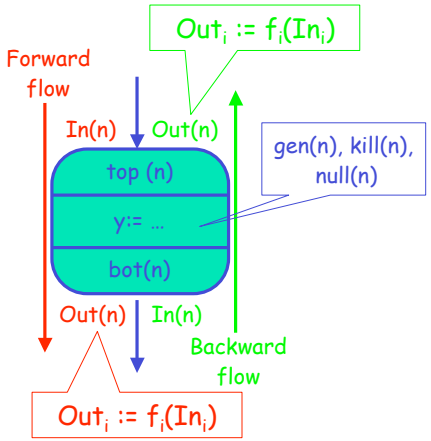
- Initial values
  - for each node define gen and kill information
- Input Equations
  - for each node we have an equation of the form:
    In_i := Merge (Out_j)
  - "Merge" operation over the "predecessors" of n_i



General Approach

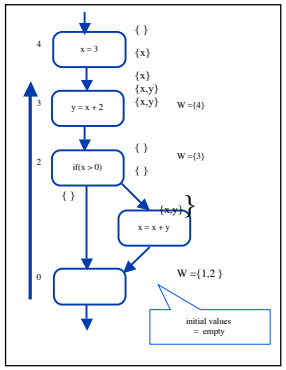
- Transfer Equations
  - for each node we have an equation of the form: $Out_i := f_i(In_i)$
  - Transfer functions usually depend on Gen/Kill information that is computed for each node
  - Usually:

    Out := (In - kill)U gen
- We can view the set of variables, transfer functions, and flow graph as a system of equations



worklist algorithm

1. Start at initial node (entry for forward; exit for reverse), label IN_0 with pertinent "facts" (initial values)
2. Compute $OUT_0 = F(IN_0)$ (label OUT_0 with the computed facts)
3. Propagate OUT_0 to IN_i (label edge $N_o \Rightarrow N_i$ with OUT_0) where N_i are successor nodes (forward) or predecessor nodes (reverse) of N_0
4. Compute $OUT_i = F(IN_i)$, place all N_i on a "worklist" W, and for all N_i label OUT_i with the computed facts.
5. While W is not empty,
   1. pick N_i from W and propagate OUT_i to IN_k (label edges $N_i \Rightarrow N_k$ with OUT_i) where N_k are successor nodes (forward) or predecessor nodes (reverse) for N_i; delete N_i from W
   2. Compute $OUT_k = F(IN_k)$ for all N_k where IN_k=MERGE all input edge labels (MERGE = ∪ for "some paths" and ∩ for "all paths"), label OUT_k with the computed facts); and if for N_k, OUT_k changes put N_k, on W'
6. If W' is not empty, then W=W' and go to 5

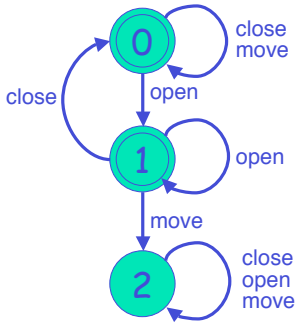


Using Quantified Regular Expressions

- Alphabet, quantification, regular expression
- For the events {open, close, move}

  show that for all paths:

  ((close v move)*,
  (open+ v open+,close))*



Cecil: Olender and Osterweil

- Instead of implicitly defined facts, let the user define application-specific facts
  - Represented as a Deterministic Finite State Automaton (DFSA) or as a Quantified Regular Expression (QRE)
  - Events
    - Recognizable events
    - Method calls
      - Can reason about sequences of method calls
      - E.g.,Push must be called before Pop
    - Thread interactions
      - Join or Fork
    - Arbitrary operations
      - a+b
    - Need to be able to treat events as indivisible actions
      - E.g., can treat pop and push as atomic as long as they do not contain any events of concern
- Propagate the states in the DFSA that can reach each node in the program

State Propagation

- States of the property are propagated through the CFG
- The property is proved if only accepting (non-accepting) states are contained in the final node of the CFG
- Cecil DFSA ->

  lattice ($\mathcal{P}$(S), $\subset$, $\cup$)

  function space

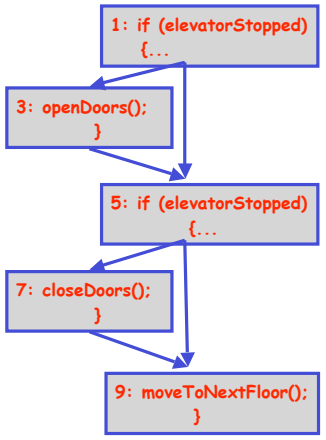
  $\delta : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$

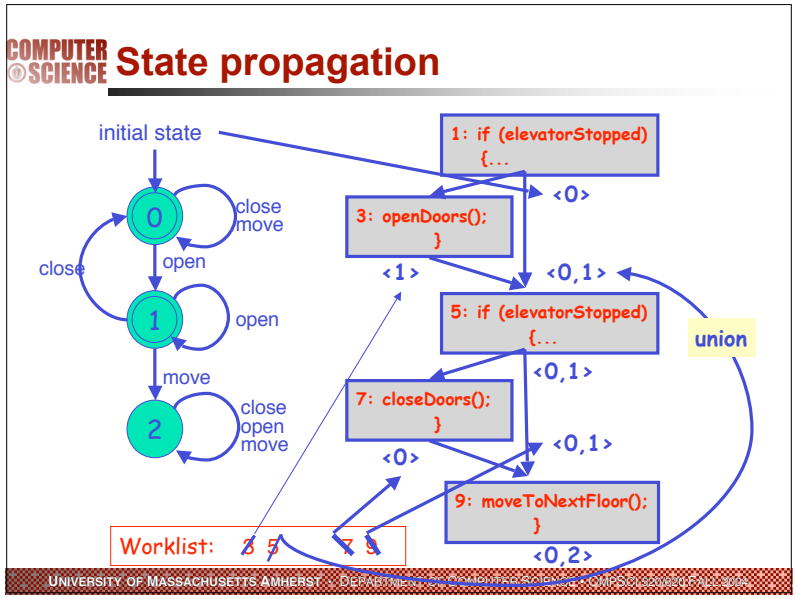
  facts at nodes are elements of $\mathcal{P}$(S)
- propagate until convergence and check if terminal node in an accepting state of DFSA

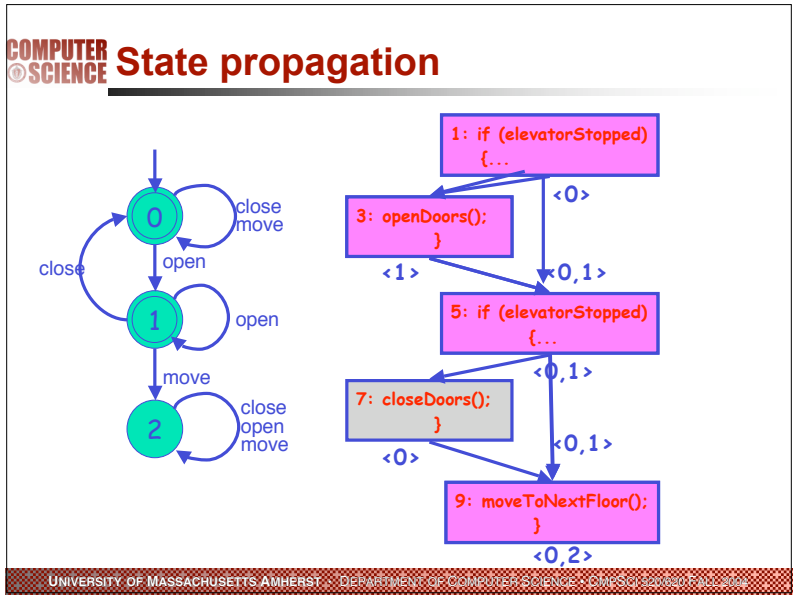
Elevator Controller

```
    void main()
    {
    ...
1:  if (elevatorStopped)
    {...
3:          openDoors();
    }
    ...
5:  if (elevatorStopped)
    {...
7:          closeDoors();
    }
9:  moveToNextFloor();
    }
```

- States of the property are propagated through the CFG
- For an **all** property: the property is proved if only accepting states are contained in the final node of the CFG
- For a **none** property: the property is proved if only non-accepting states are contained in the final node of the CFG

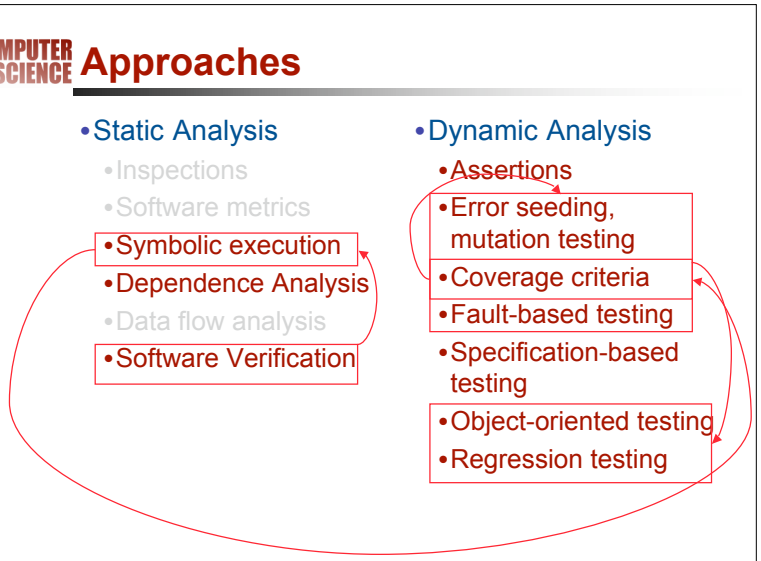






Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing



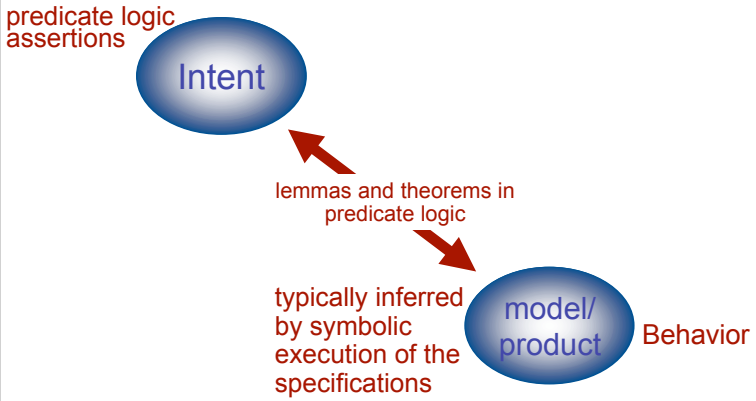
Verification

- two well-established approaches
  - (Automated) mathematical reasoning
    - theorem proving
    - proof checking
  - Finite-state verification
    - model checking
      - Logic spec + FSA comp model ⇒ symbolic model checking
      - FSA spec + FSA comp model ⇒ automata-theoretic model checking
    - property checking

Verification

- How are they different?
  - (Automated) mathematical reasoning
    - difficult, error prone
    - decidability vs. expressiveness
      - Propositional calculus is decidable
      - Predicate calculus is semi-decidable
  - Finite-state verification
    - Reason about a finite model of the system
    - Fast, yields counterexamples, manages partial specifications, applies to concurrency
    - State explosion!

Proof



Floyd Method of Inductive Assertions

- Show that given the input assertions, after executing the program, program satisfies output assertions
  - show that each program fragment behaves as intended
  - use induction to prove that all fragments, including loops, behave as intended
- show that the program must terminate
- informal description
  - Place assertions at the start, final, and intermediate points in the code.
  - Any path is composed of sequences of program fragments that start with an assertion, are followed by some assertion free code, and end with an assertion
    - $A_s, C_1, A_2, C_2, A_3, \ldots A_{n-1}, C_{n-1}, A_f$
  - Show that for **every** executable path, if A_s is assumed true and the code is executed, then A_f is true

Why does this work?

- suppose P is an arbitrary path through the program
- can denote it by

$$P = A_0\, C_1\, A_1\, C_2\, A_2 \ldots C_n\, A_n$$

- where
  - A_0 - Initial assertion
  - A_n - Final assertion
  - A_i - Intermediate assertions
  - C_i - Loop free, uninterrupted, straight-line code

If it has been shown that

$$\forall\, i,\ 1 \le i < n\text{: } A_i C_i \Rightarrow A_{i+1}$$

Then, by transitivity

$$A_0 \Rightarrow \ldots\ldots \Rightarrow A_n$$

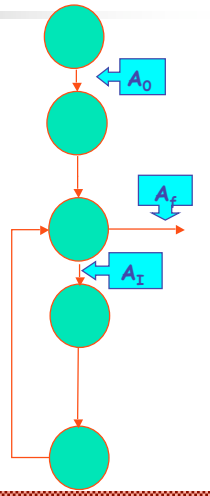
Obvious problems

- How do we do this for a path?
- How do we do this for all paths?
  - Infinite number of paths
    - Must find a way to deal with loops

Find loop invariant (A_I)

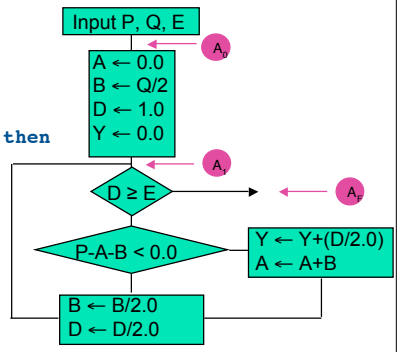
- subpaths to consider:
  - C_1 Initial assertion A_0 to final assertion A_f
  - C_2 Initial assertion A_0 to A_I
  - C_3 A_I to A_I
  - C_4 A_I to final assertion A_f
- Basically an inductive proof

The "Aha!" moment - finding invariants is hard!



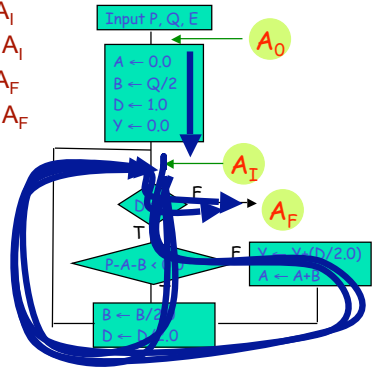
Wensley's Algorithm

```
Procedure Wensley (P:input, Q:input, E:input, Y:output);
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;
```



Floyd Proof: Wensley's Algorithm

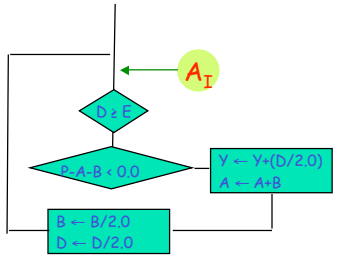
- Summary of Five Lemmas Needed
  - A_0 to A_I
  - A_I, true branch, to A_I
  - A_I, false branch, to A_I
  - A_I, true branch, to A_F
  - A_I, false branch, to A_F



Lemma III: AI, false branch, to AI

A_I: {(A=Q*Y)∧(B=Q*(D/2))
∧ (k ≥ 0, k integer ∧ D=2^{-k})
∧ ((P/Q)-D)<Y≤(P/Q)}

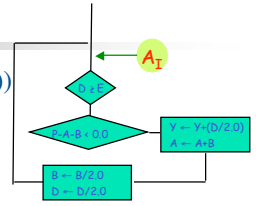
code
- D ≥ E [constraint]
- P - A - B ≥ 0 [constraint]
- Y ¨ Y+(D/2.0)
- A ¨ A+B
- B ← B/2
- D D/2



A'$_I$: {(A'=Q*Y')∧(B'=Q*(D'/2))
∧ (k ≥ 0, k integer ∧ D'=2^{-k})
∧ ((P/Q)-D')<Y'≤(P/Q)}

proof of lemma III

$A_I \Rightarrow A'_I$: {(A'=Q*Y') ∧ (B'=Q*(D'/2))
∧ (k ≥ 0, k integer ∧ D'=2^{-k})
∧ ((P/Q)-D')<Y'≤(P/Q)}



we have

A' = A + B; B' = B/2.0; D' = D/2.0; Y' = Y + D/2.0;

1) A' = A+B = Q*Y + Q*(D/2) = Q*(Y+(D/2));
Y'= Y+(D/2); ∴A' = Q * Y'

2) B'= B/2 =(Q *D/2)/2; D'=D/2
∴B' = (Q * 2D'/2)/2 = Q* D'/2

from A_I

and so on … basically using symbolic evaluation

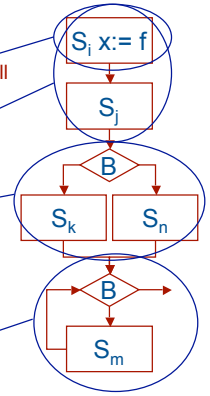
Hoare axiomatic proof

- assertions are preconditions and post conditions on some statement or sequence of statements

  P{S}Q
- if P is true before S is executed and S is executed then Q is true
- as in Floyd's inductive assertion method, we construct a sequence of assertions, each of which can be inferred from previously proved assertions and the rules and axioms about the statements and operations of the program
- to prove P{S}Q, we need some axioms and rules about the programming language

Hoare axioms and proof rules

- axiom of assignment
  - P {x:=f} Q,
  - where Q is obtained from P by substituting f for all occurrences of x in P (symbolic execution)
- rule of composition
  - P {S1, S2 } Q => ∃ P1 , P{S1}P1 ∧ P1{S2}Q
- rule for the alternative statement
  - P{if B then S1 else S2 }Q ⇒ P{B ∧ S1}Q ∧ P{¬B ∧ S2}Q
- rules of consequence
  - [P {S} Q ∧ Q ⇒ R] ⇒ P {S} R
  - [P {S} Q ∧ R ⇒ P] ⇒ R {S} Q
- rule of iteration
  - P {while B do S }Q ⇒ P{~B}Q ∧ ∃ I ⇒ P {B ∧ S} I ∧ I{B ∧ S } I ∧ I{~B }Q



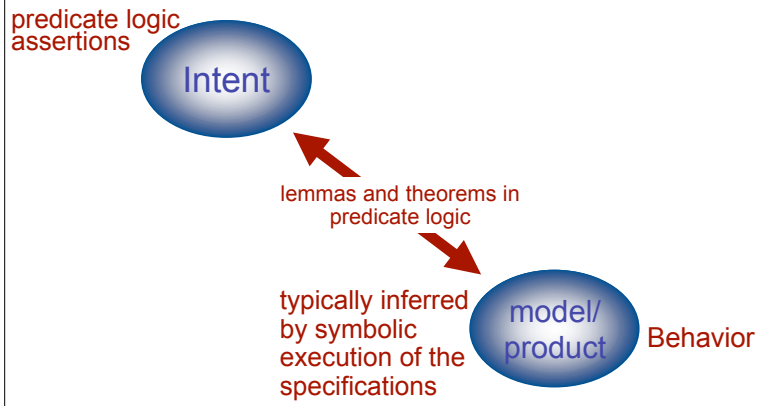
loop invariant

backwards substitution

Proof

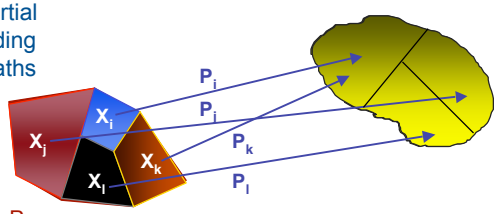
- Hoare-style and Floyd- style verification are essentially the same
  - one is based on graphical representation and the other on a textual representation.
    - In Floyd-style proof, we visualize the proof goal by annotating a CFG
    - In the other, we define the proof goal as a Hoare triple
- Mechanism for applying proof
  - may work either direction on such a proof, but because it's typically easier to work backwards, often use a technique called backwards substitution
  - we work our way from the post-condition, using the proof rules to "push formulas through" the program
  - at each point where a "pushed-through" predicate "runs into" a supplied predicate, we have a verification condition (VC) that must be proved.
  - After all VCs are proved, we need to be prove termination
    - Without a termination proof, we achieve **partial correctness**
    - With a termination proof, we achieve **total correctness**

Proof



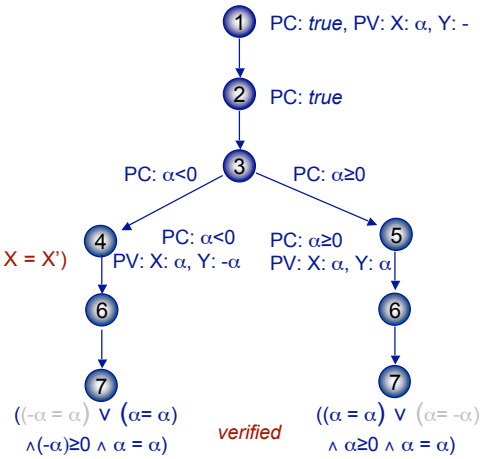
Symbolic Evaluation/Execution

- Creates a functional representation of a path of an executable component
- P is composed of partial functions corresponding to the executable paths
  $P = \{P_1,...,P_r\}$
  $P_i : X_i \rightarrow Y$
- For a path P_i
  - $D[P_i]$ is the domain for path P_i
  - $C[P_i]$ is the computation for path P_i

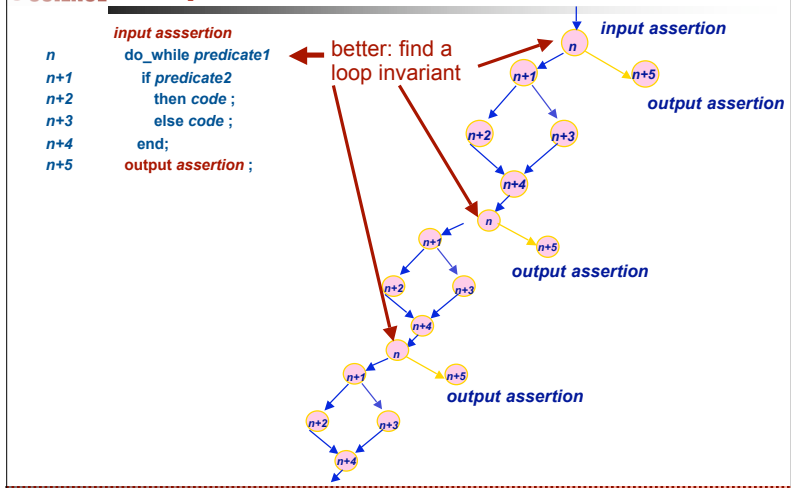


Execution tree (Hantler-King)

```
ABSOLUTE
assume (true)
1   procedure(X);
2   declare X,Y integer
3   if X<0
4       then Y← -X;
5       else Y← X;
6   return (Y);
prove((Y = X'|Y = -X') & Y≥0 & X = X')
7   end;
```



Loops -- unroll them?



Straightforward Observations

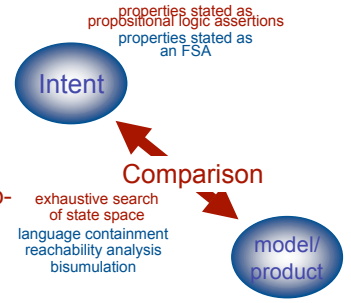
- Problems
  - formal proofs are long, tedious and are often hard; assertions are hard to get right; invariants are difficult to get right (need to be invariant, but also need to support overall proof strategy)
- Unsuccessful proof attempt ⇒ ???
  - incorrect software? assertions? placement of assertion? inept prover? although failed proofs often indicate which of the above is likely to be true (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 ⇒ errors in the proof are more likely than errors in the software being verified

Verification

- How are they different?
  - (Automated) mathematical reasoning
    - difficult, error prone
    - decidability vs. expressiveness
      - Propositional calculus is decidable
      - Predicate calculus is semi-decidable
  - Finite-state verification
    - Reason about a finite model of the system
    - Fast, yields counterexamples, manages partial specifications, applies to concurrency
    - State explosion!

Model Checking: Overview

- properties usually expressed in
  - in a propositional logic (e.g., temporal logic)
  - as a FSA
- system represented as a (possibly "abstracted") reachability graph
- reasoning engine
  - logic ⇒ propagates valid sub-formulas through the graph
  - FSA ⇒ compares FSAs via language inclusion; reachability; or bisumulation



Conservative Analysis

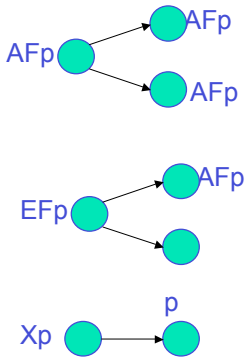
- If property is verified, property holds for all possible executions of the system
- If property is not verified:
  - an error found
    OR
  - a spurious result
- System model abstracts information to be tractable
  - Conservative abstractions usually over-approximate behavior
  - If inconsistency relies upon over-approximations, then a spurious result
  - e.g. all counter example correspond to infeasible paths

Temporal logic

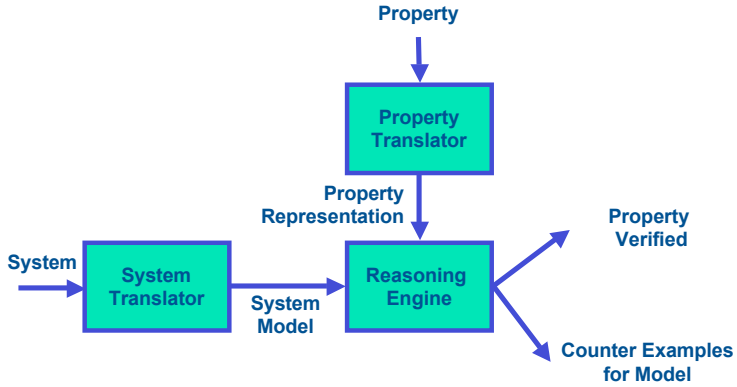
- augments the standard operators of propositional logic with “tense” operators
- "possible worlds semantics" ⇒ Kripke model
  - relativize the truth of a statement to temporal stages or states
  - a statement is not simply true, but true at a particular state
  - states are temporally ordered, with the type of temporal order determined by the choice of axioms.
- model of time
  - partially ordered time
  - linearly ordered time
    - linear temporal logic is typically extended by two additional operators, “until” and “since”
  - discrete time
  - branching (nondeterministic) time
    - foundation for one of the principal approaches to verifying concurrent systems = Computational Tree Logics.

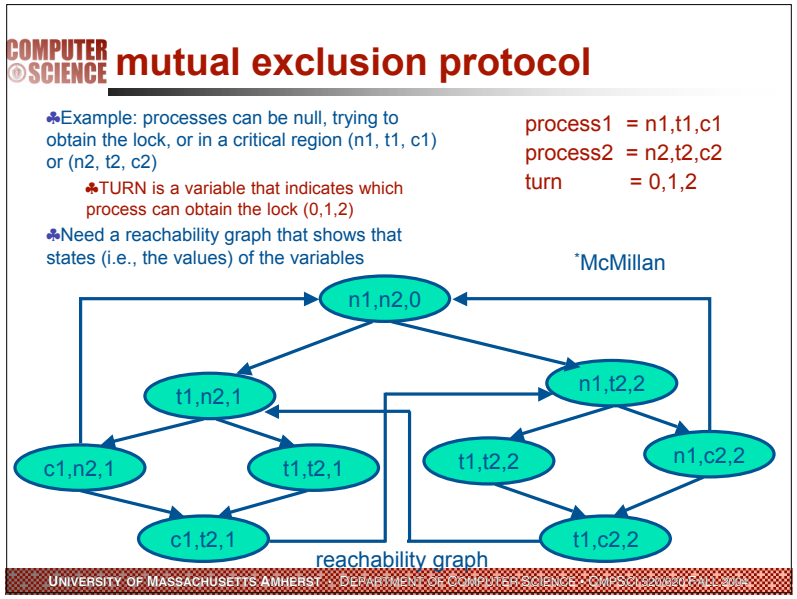
Computation Tree Logics

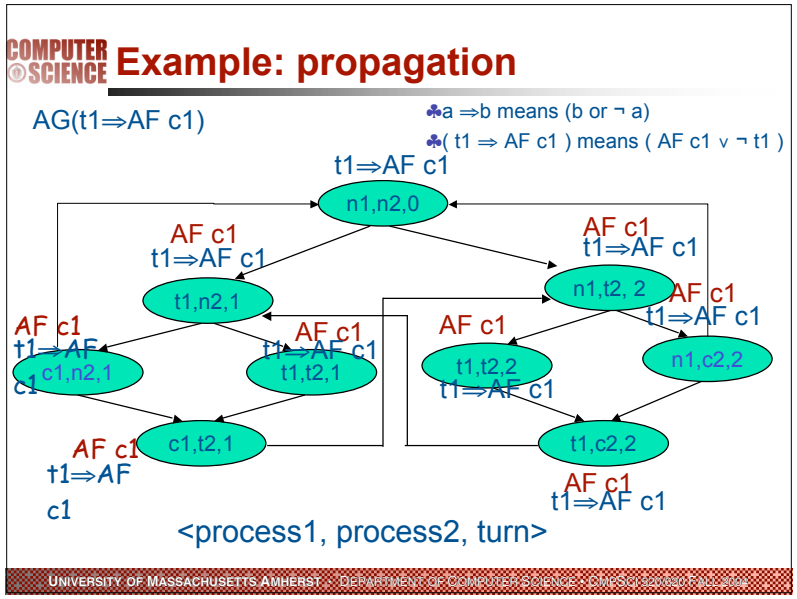
- specification language
  - a propositional temporal logic.
- verification procedure
  - exhaustive search of the state space of the concurrent system to determine truth of specification.
- formulas constructed from path quantifiers and temporal operators:
  - path quantifier:
    - A “for every path”
    - E “there exists a path”
  - temporal operator:
    - Xp “p holds next time’
    - Fp “p holds sometime in the future”
    - Gp “p holds globally in the future”
    - pUq “p holds until q holds”



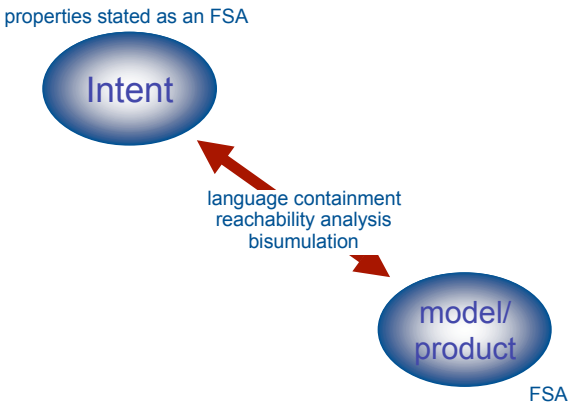
Architecture of FSV Systems





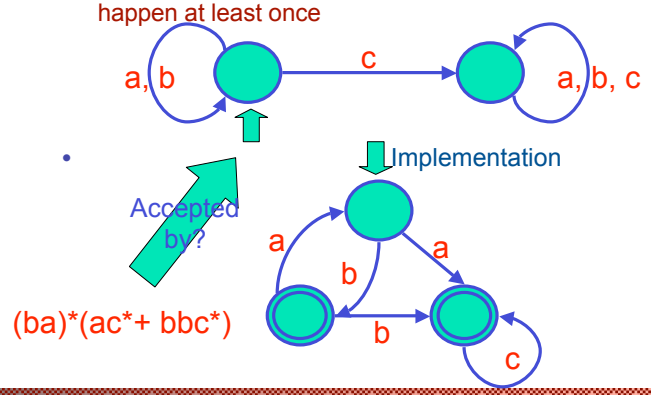


Automata-Theoretic Model Checking



Example

- Specification:
  - of the possible observable events (a, b, c), c must happen at least once



Some observations

- Model Checking
  - worst case bound linear in size of the model
    - but the model is exponential
  - not clear if model checking or symbolic model checking is superior
    - depends on the problem
  - experimentally often very effective!
    - used selectively to verify hardware designs
    - trying to develop appropriate abstractions to make it applicable to software systems

Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

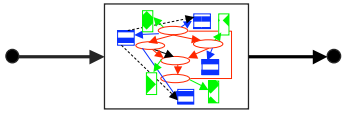
Types of Testing--what is tested

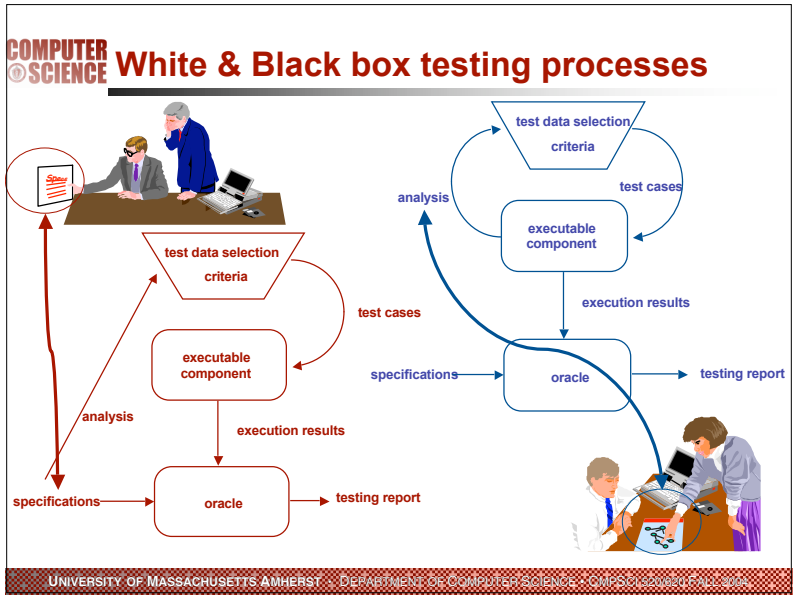
- Unit testing
  - exercise a single simple (procedure) component
- Integration testing
  - exercise a collection of inter-dependent components
  - focus on interfaces between components
- System testing
  - exercise a complete, stand-alone system
- Acceptance testing
  - customer's evaluation of a system
  - usually a form of system testing
- Regression testing
  - exercise a changed system
  - Focus on modifications or their impact

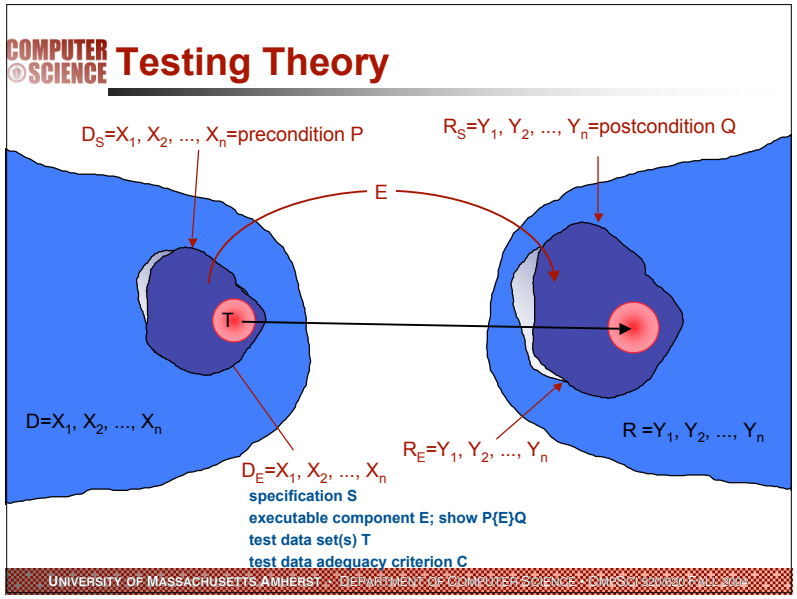
Testing approaches

- "black box"

- "white box" or "glass box"







Testing Theory

- Criterion C for Test Adequacy
  - C:SxE ->2^T
  - specification-based C (s) — "black box"
  - interface-based C (x1, x2, ..., xn, y1, y2, ..., yn)
  - program-based C (e)
  - combined C (s,e) — "white box"
- Types
  - structural
  - fault-based
  - error-based
- if specification S defines a function F, such that P{F}Q, then C is *reliable* if $T_1, T_2, \ldots, T_m$; $C(T_i, E)$; and $D(E) \supseteq T_i$
  - $\forall T_i (\forall t \in T_i, E(t) = F(t)) \vee \forall T_i (\exists t \in T_i \; E(t) \neq F(t))$
  - $\forall t \in T_i, E(t) = F(t) \Rightarrow \forall t \in T_i \; OK(T_i) \Rightarrow E \equiv F$

Ideal Test Criterion

- test criterion C is *ideal* if for any executable component E and every test set $T_i \subseteq D(E)$ such that $C(T_i, E)$, T_i is successful
  - of course we want T_i << D(E)
  - but in general, T= D(P) is the only ideal test criterion
- In general, there is no ideal test criterion
  - "Testing shows the presence, not the absence of bugs" E. Dijkstra
- Dijkstra was arguing that verification was better than testing
- but, verification has similar problems
  - can't prove an arbitrary program is correct
    - can't solve the halting problem
  - can't determine if the specification is complete
- need to use these techniques so that they compliment one another

Black Box Testing

- Functional/Interface Test Data Selection
  - typical cases
  - boundary conditions/values
  - illegal conditions (if robust)
  - fault-revealing cases
    - based on intuition about what is likely to break the system
  - other special cases
- stress testing
  - large amounts of data
  - worse case operating conditions
- combinations of events
  - select those cases that appear to be more error-prone
- common representations for selecting sequences of events
  - decision tables
  - cause and effect graphs
  - usage scenarios

"White Box" Test Data Selection

- structural
  - coverage based
- fault-based
  - e.g., mutation testing, RELAY
- error-based
  - domain and computation based
  - use representations created by symbolic execution