COMPUTER SCIENCE

24- Static Analysis

Rick Adrion

COMPUTER SCIENCE

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

COMPUTER SCIENCE

Review methods

- Fagan inspections
  - formal, multi-stage process
  - significant background & preparation
  - led by moderator
- Active design reviews
  - also called "phased inspections"
  - several brief reviews rather than one large review
  - guided by questions from the author
- Cleanroom
  - more than reviews, but reviews important component
  - we'll come back to this
- N-fold
  - parallel reviews controlled by moderator
  - focuses on user requirements

COMPUTER SCIENCE

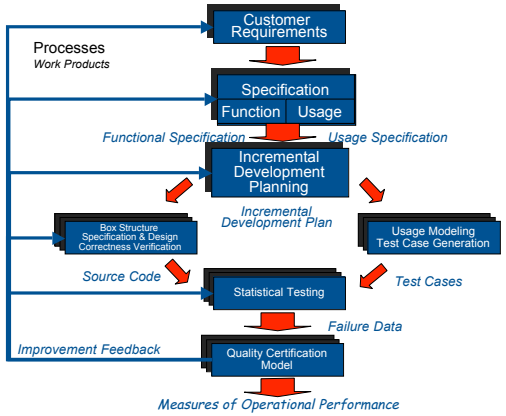
Why are inspections effective

- knowing the product will be scrutinized causes developers to produce a better product
- having others scrutinize a product increases the probability that faults will be found
- walkthroughs and reviews are not as formal as inspections, but appear to also be effective
  - hard to get empirical results

What are the deficiencies?

- focus on error detection
  - what about other "ilities" -- maintenance, portability, etc.
- not applied consistently & rigorously
  - inspection shows statistical improvement, but cannot ensure quality
  - inspection should have the same results without regard to the product to which it is applied or the inspection team
- range of errors not addressed
  - team expertise limited
  - one property may have many error modalities
- human intensive and often makes ineffective use of human resources
  - e.g., skilled software engineer reviewing coding standards, comments spelling, etc.
- no automated support
  - again inefficient of human resources
- aspects of review not used appropriately
  - e.g., in Fagan process, overview often covers what should be described if documentation is adequate

Cleanroom

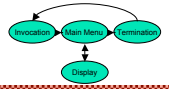


Cleanroom

```
[f]
do
     [g]
     [h]
od

For all
inputs, does
[g] followed
by [h] do
[f]?
```

- Verification as Review Process
  - team verification of correctness takes the place of individual unit testing; correctness is established by **group consensus** if it is obvious
  - by formal proof techniques if it is not.
- benefits
  - intellectual control of the process
  - motivates developers to deliver error-free code
  - verification is a form of peer review
  - each person assumes responsibility for and derives a sense of ownership in the evolving product
- every person must agree that the work is correct before it is accepted -> successes are ultimately team successes, and failures are team failures.
- Markov Analysis
- Factors
  - number of statistically typical (i.e., likely) usage paths through the software
- Steps
  - focus verification efforts,
  - identify the likelihood of given events,
  - project the test schedule, and
  - ascertain the (affordable) upper bound on inferences about reliability
- Stopping Criterion for Testing
  - goals (e.g., target level of estimated reliability) are achieved
  - or quality standards (e.g., errors/KLOC) are violated



Software Metrics

- measures that predict qualities about software
- can be applied to any of the products (e.g., design, code, test cases) or to the process (e.g., Capability Maturity Model)
- Qualities measured by software metrics
  - performance
  - user-friendliness
  - resources
    - memory/storage
    - development costs
    - maintenance cost
  - quality
    - maintainabity
    - reliability
    - completeness
    - consistency
    - complexity

Function Points

- proposed by Albrecht in 1979
  - Originally applied to code
- UFP =
  number of inputs x w1 +
  number of outputs x w2 +
  number of user inquiries x w3
  number of files x w4 +
  number of external references x w5
- function points = UFP* TCA = UFP* (.65 + 0.01 * SUM(Fi))
  - where the degree of influence, DI= SUM(Fi) is the sum of complexity adjustment values, Fi
- metrics:
  - productivity: FP/person-month
  - quality: defects/FP
  - cost: $/FP

- weights:

	Simple	Average	Complex
w1	3	4	6
w2	3	5	7
w3	3	4	6
w4	7	10	15
w5	5	7	10

More Quality Metrics

- Modularity
  - cohesion metric
    - applied to unit design
    - the relationship among the elements of a module
    - best cohesion level is functional, and the worst is coincidental.
  - Cruickshank and Gaffney Cohesion Strength
    Strength = $\sqrt{(X^2 + Y^2)}$
    - where:
    - X = reciprocal of the number of assignment statements in the module
    - Y = number of unique function outputs divided by number of unique function inputs

More Quality Metrics

- Modularity
  - coupling
    - applied to system and unit designs
    - measure of the degree to which modules share data
    - data coupling (the sharing of data via parameter lists) is the best type of coupling, while common coupling (the sharing of data via global or common areas) is the worst.
    - a lower coupling value is better.
  - Cruickshank and Gaffney Coupling:
    - M_i = sum of the number of input and output items shared between components i & j
    - Z_i = average number of input and output items shared over m components with component i
    - n = number of components in the software product

$$Coupling = \frac{\sum_{i=1}^{n} Z_i}{n}$$

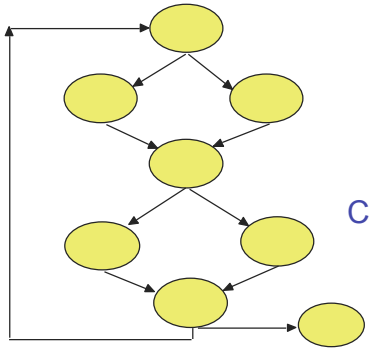
where:

$$Z_i = \frac{\sum_{j=1}^{m} M_i}{m}$$

McCabe's cyclomatic complexity

- Complexity measured by control flow information
  - based on a control flow graph where e is number of edges, n is number of nodes, p is number of connected components
- McCabe's Cyclomatic Complexity:
  - v = e - n + 2
    - where:
    - v = complexity of the graph
    - e = number of edges (program flows between nodes)
    - n = number of nodes (sequential groups of program statements)
  - if a strongly connected graph is constructed (one in which there is an edge between the exit node and entry node), the calculation is
  - v = e - n + 1

Example



n = 8
e = 10
p = 1

C = 10 - 8 + 2 = 4

Software Science

- Halstead applied information theory to computer science
- metrics
  - n_1 number of distinct operators
  - n_2 number of distinct operands
  - N_1 total number of occurrences of operators
  - N_2 total number of occurrences of operands
- program level estimator

$$\mathcal{D} = 1/\mathcal{L} = (n_1/2)(N_2/n_2)$$

$$\mathcal{L} = 1/\mathcal{D} = (2/n_1)(n_2/N_2)$$

difficulty increases as operators are introduced (n_1/2 increases) and as operands are used repetitively (N_2/n_2 increases)

- programming time

$$\mathcal{T} = \mathcal{E}/\mathcal{S}$$

where $\mathcal{S}$ is the "Stroud number"

$5 \le \mathcal{S} \le 20$, usually 18

Software Science (continued)

- language level

$$\lambda = \mathcal{L} \times \mathcal{V}^* = \mathcal{L}^2\mathcal{V}^*$$

$\lambda_{PL/1} = 1.53$, $\lambda_{Algol} = 1.21$,
$\lambda_{Fortran} = 1.14$, $\lambda_{CDC\ assmblr} = 0.88$

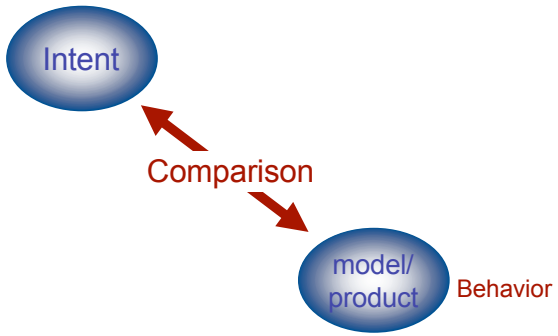
- predicted effort

$$\mathcal{E} = \mathcal{V}^{*3}/\lambda^2$$

Quality Metrics for Code

- Understandability
  - size metrics
    - lines of code
    - function points
    - function count
  - traceability metrics
    - number of comment lines per total source lines of code
    - percent comment lines of total lines
    - correctness of comments
- Predicting quality
  - LOC X domain seems to be the most reliable predictor

Analysis



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

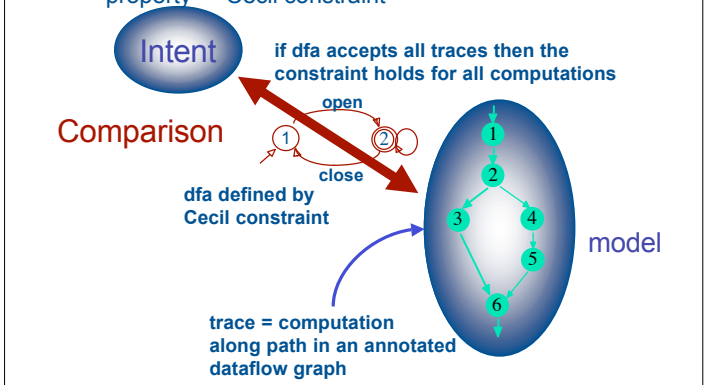
Compare behavior to intent

- 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



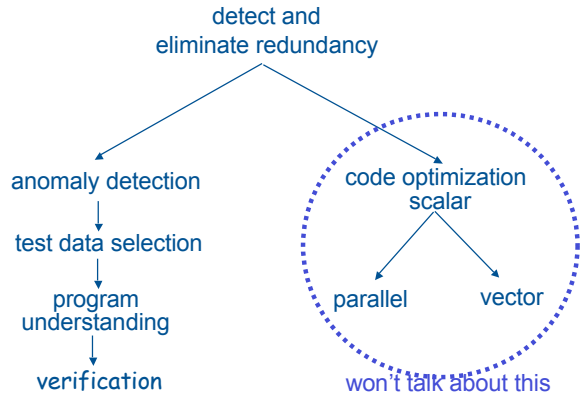
Data Flow Analysis (DFA)

- 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

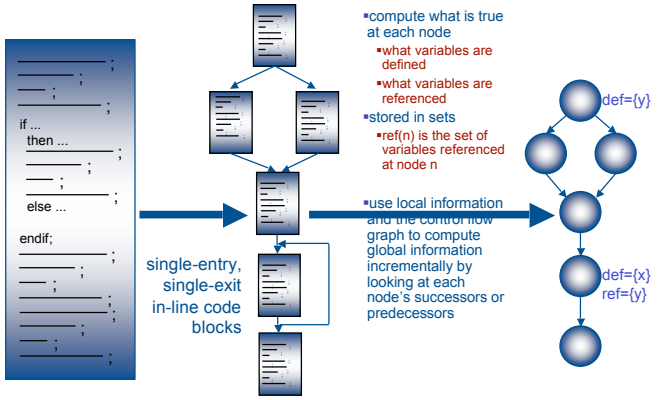
Data flow analysis

- 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

Data Flow Analysis



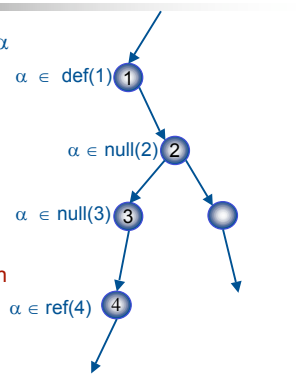
DFA



- compute what is true at each node
  - what variables are defined
  - what variables are referenced
- stored in sets
  - ref(n) is the set of variables referenced at node n
- use local information and the control flow graph to compute global information incrementally by looking at each node's successors or predecessors

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
  - $\alpha \in$ def (n) or
  - $\alpha \in$ ref(n) or
  - $\alpha \in$ null (n)
  - for each node n on the path
- write (and simplify) a path expression
  - $P(n_1, n_1, \ldots, n_1; \alpha)$



$P(1,2,3,4; \alpha) = d11r = dr$

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

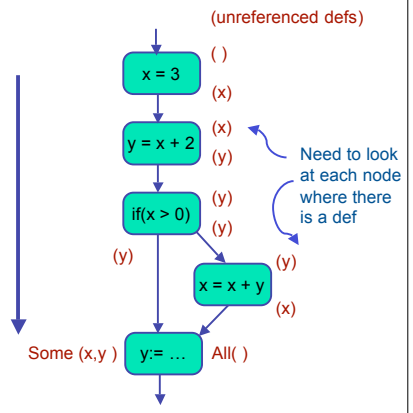
global dataflow analysis

- 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

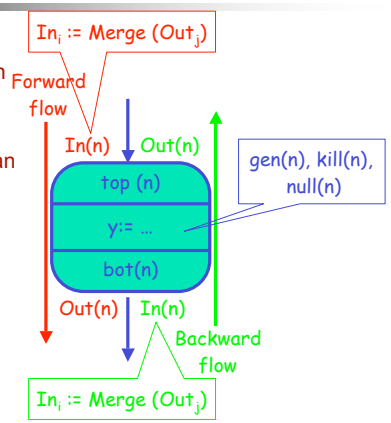
```
int x,y;
...
x := 3;
y := x + 2;
if x > 0 then
  x := x + y;
end if;
y:= …
```

Forward flow, all paths problem



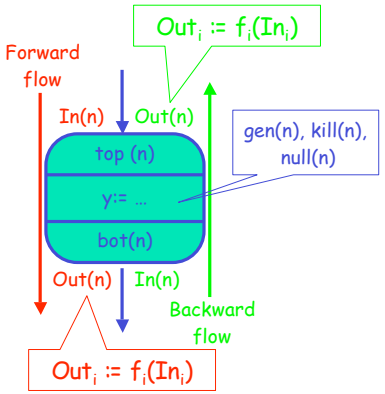
General Approach

- 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



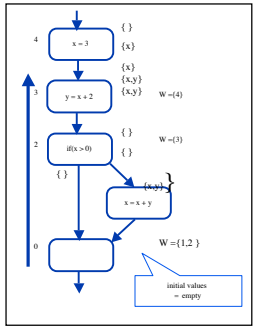
General Approach

- 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 $N_0 \Rightarrow N_i$ 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 $OUT_i = F(IN_k)$ for all N_k where IN_k=MERGE all input edge labels (MERGE = $\cup$ for "some paths" and $\cap$ 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

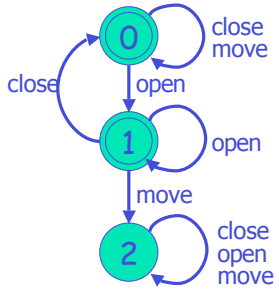


Using Quantified Regular Expressions

- 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), $\subset$, $\cup$)

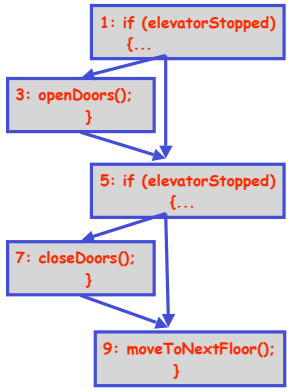
  function space

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

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

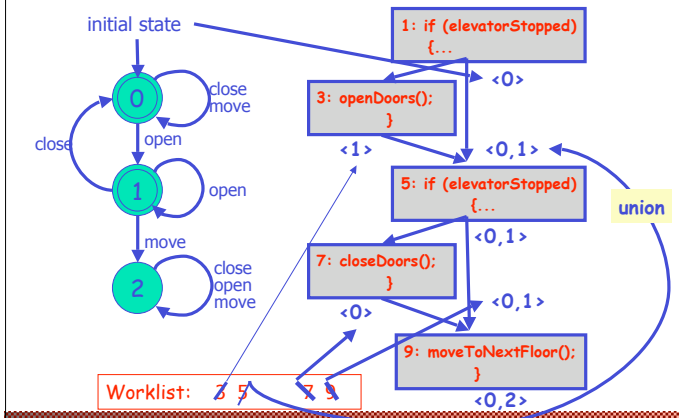
Elevator Controller

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

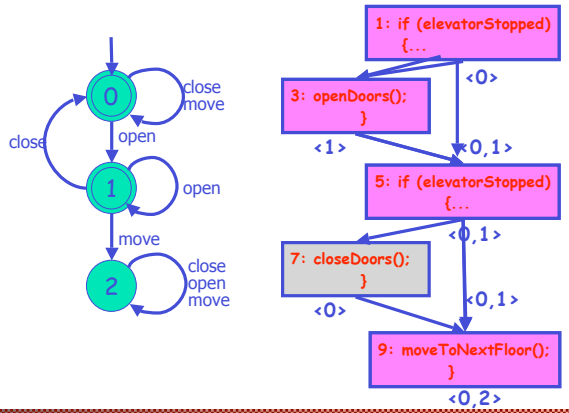


- States of the property are propagated through the CFG
- For an **all** property: the property is proved if only accepting states are contained in the final node of the CFG
- For a **none** property: the property is proved if only non-accepting states are contained in the final node of the CFG

State propagation



State propagation



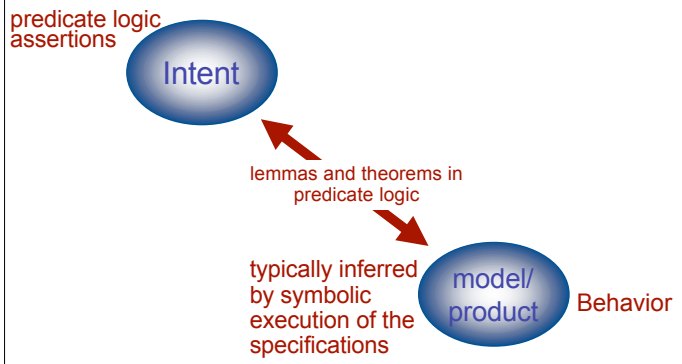
Verification

- 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

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

Then, by transitivity

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

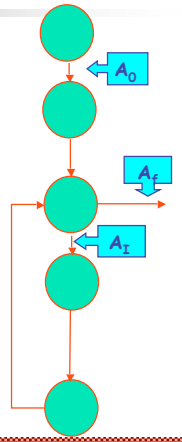
Obvious problems

- 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)

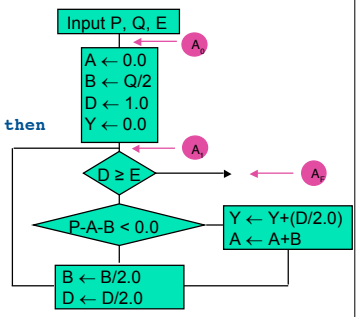
- subpaths to consider:
  - C_1 Initial assertion A_0 to final assertion A_f
  - C_2 Initial assertion A_0 to A_I
  - C_3 A_I to A_I
  - C_4 A_I to final assertion A_f
- Basically an inductive proof

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



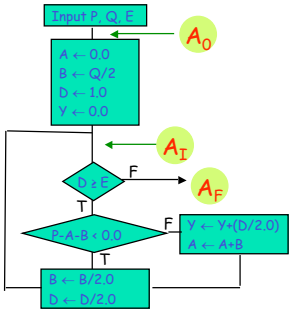
Wensley's Algorithm

```
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;
```



Summary of Five Lemmas Needed

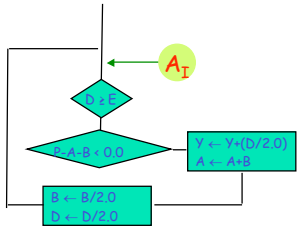
- A_0 to A_I
- A_I, true branch, to A_I
- A_I, false branch, to A_I
- A_I, true branch, to A_F
- A_I, false branch, to A_F



Lemma III: AI, false branch, to AI

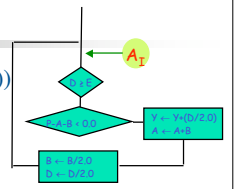
A_I: {(A=Q*Y)∧(B=Q*(D/2))
∧ (k ≥ 0, k integer ∧ D=2^{-k})
∧ ((P/Q)-D)<Y≤(P/Q)}

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

- take a simple programming language that deals only with integers and has the following types of constructs:
  - assignment statement
    x:= f
  - composition of a sequence of statements
    S1, S2
  - conditional (alternative statements)
    if B then S1 else S2
  - iteration
    while B do S

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 fi
    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

Proof

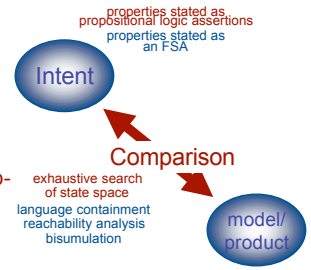
- Hoare-style and Floyd- style verification are essentially the same
  - one is based on graphical representation and the other on a textual representation.
    - In Floyd-style proof, we visualize the proof goal by annotating a CFG
    - In the other, we define the proof goal as a Hoare triple
- Mechanism 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**

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

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

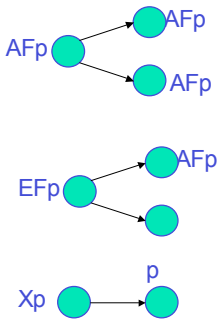
- If property is verified, property holds for all possible executions of the system
- If property is not verified:
  - an error found
    OR
  - a spurious result
- System model abstracts information to be tractable
  - Conservative abstractions usually over-approximate behavior
  - If inconsistency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

Temporal logic

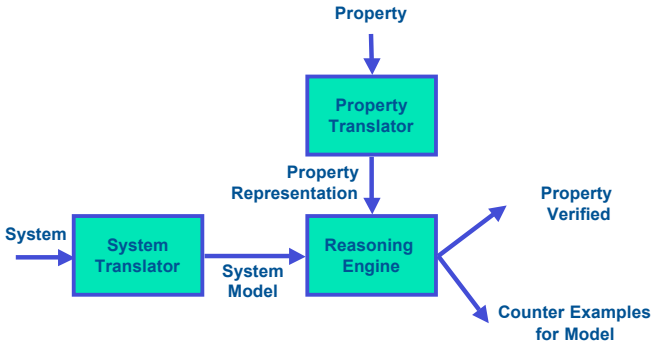
- augments the standard operators of propositional logic with "tense" operators
- "possible worlds semantics" ⇒ Kripke model
  - relativize the truth of a statement to temporal stages or states
  - a statement is not simply true, but true at a particular state
  - states are temporally ordered, with 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

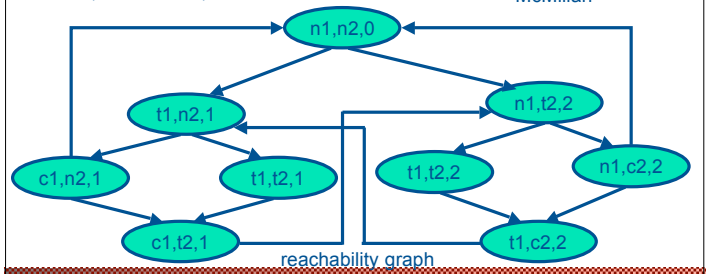


mutual exclusion protocol

- Example: processes can be null, trying to obtain the lock, or in a critical region (n1, t1, c1) or (n2, t2, c2)
  - TURN is a variable that indicates which process can obtain the lock (0,1,2)
- Need a reachability graph that shows that states (i.e., the values) of the variables

process1 = n1,t1,c1
process2 = n2,t2,c2
turn = 0,1,2

*McMillan

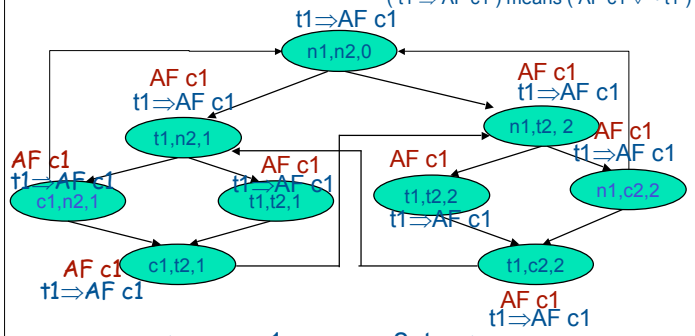


reachability graph

Example: propagation

AG(t1⇒AF c1)

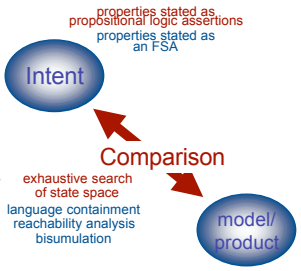
- a ⇒b means (b or ¬ a)
- (t1 ⇒ AF c1) means (AF c1 ∨ ¬ t1)



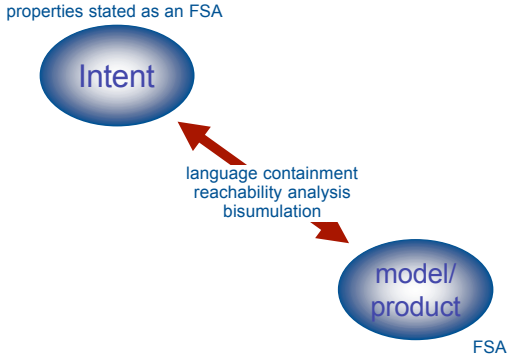
<process1, process2, turn>

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

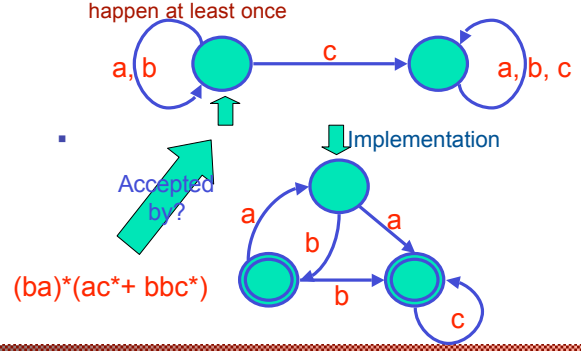


Automata-Theoretic Model Checking



Example

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



- Implementation

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